- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Introduction

October 1, 1999: Book is on Delphi 5 Companion CD.
Source Code available. Examples list added.

Book Cover. Apollo, the god worshipped at Delphi,
in an Italian 17th century fresco.

The first few editions of Mastering Delphi, the best selling Delphi book I've written, provided an introduction to the Pascal language in Delphi. Due to space constraints and because many Delphi programmers look for more advanced information, in the latest edition this material was completely omitted. To overcome the absence of this information, I've started putting together this online book, titled **Essential Pascal**.

This is a detailed book on Pascal, which for the moment will be available for free on my web site (I really don't know what will happen next, I might even find a publisher). This is a work in progress, and any feedback is welcome. The first complete version of this book, dated July '99, has been published on the Delphi 5 Companion CD.

Copyright

The text and the source code of this book is copyrighted by Marco Cantù. Of course you can use the programs and adapt them to your own needs, only you are not allowed to use them in books, training material, and other copyrighted formats. Feel free to link your site with this one, but please do not duplicate the material as it is very subject to frequent changes and updates.

The Book Structure

The following is the current structure of the book:

Source Code

The source code of all the examples mentioned in the book is available. The code has the same Copyright as the book: Feel free to use it at will but don't publish it on other documents or site. Links back to this site are welcome.

Download the source code in a single zip file, **EPasCode.zip** (only 26 KB in size) and check out the **list of the examples**.

Feedback

Please let me know of any errors you find, but also of topics not clear enough for a beginner. I'll be able to devote time to the project depending also on the feedback I receive. Let me know also which other topics (not covered in Mastering Delphi 4) you'd like to see here. Again, hook onto the newsgroup, listed on my web site, and look for the books section, or mail to **marco@marcocantu.com** (putting **Essential Pascal** in the subject (and your request or comment in the text).

Acknowledgements

If I'm publishing a book on the web for free, I think this is mainly due to Bruce Eckel's experience with Thinking in Java. I'm a friend of Bruce and think he really did a great job with that book and few others.

As I mentioned the project to people at Borland I got a lot of positive feedback as well. And of course I must thank the company for making first the Turbo Pascal series of compilers and now the Delphi series of visual IDEs.

I'm starting to get some precious feedback. The first readers who helped improving this material quite a lot are Charles Wood and Wyatt Wong. Mark Greenhaw helped with some editing the text. Rafael Barranco-Droege offered a lot of technical corrections and language editing. Thanks.

Author

Marco Cantù lives in Piacenza, Italy. After writing C++ and Object Windows Library books and articles, he delved into Delphi programming. He is the author of the Mastering Delphi book series, published by Sybex, as well as the advanced Delphi Developers Handbook. He writes articles for many magazines, including The Delphi Magazine, speaks at Delphi and Borland conferences around the world, and teaches Delphi classes at basic and advanced levels.

You can find more details about Marco and his work on his web site, **www.marcocantu.com**.

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Appendix B: Examples

This is a list of the examples which are part of Essential Pascal and available for download:

Chapter 3

ResStr: resource strings
Range: ordinal types ranges
TimeNow: time manipulation

Test

Chapter 4

GPF: general protection faults with null pointers

Chapter 5

IfTest: if statements
Loops: for and while statements

Chapter 6

OpenArr: open array parameters
DoubleH: simple procedures
ProcType: procedural types
OverDef: overloading and default parameters

Chapter 7

StrRef: strings reference counting
LongStr: using long strings
FmtTest: formatting examples

Chapter 8

DynArr: dynamic arrays
WHandle: Windows handles
Callback: Windows callback functions
StrParam: command line parameters

Chapter 10

VariTest: simple variant operations
VariSpeed: the speed of variants

- **www.marcocantu.com**
- **Marco's Delphi Books**
- **Essential Pascal - Web Site**
- **Essential Pascal - Local Index**

Marco Cantù's Essential Pascal

Chapter 1 Pascal History

The Object Pascal programming language we use in Delphi wasn't invented in 1995 along with the Borland visual development environment. It was simply extended from the Object Pascal language already in use in the Borland Pascal products. But Borland didn't invent Pascal, it only helped make it very popular and extended it a little...

This chapter will contain some historical background on the Pascal language and its evolution. For the moment it contains only very short summaries.

Wirth's Pascal

The Pascal language was originally designed in 1971 by Niklaus Wirth, professor at the Polytechnic of Zurich, Switzerland. Pascal was designed as a simplified version for educational purposes of the language Algol, which dates from 1960.

When Pascal was designed, many programming languages existed, but few were in widespread use: FORTRAN, C, Assembler, COBOL. The key idea of the new language was order, managed through a strong concept of data type, and requiring declarations and structured program controls. The language was also designed to be a teaching tool for students of programming classes.

Turbo Pascal

Borland's world-famous Pascal compiler, called Turbo Pascal, was introduced in 1983, implementing "Pascal User Manual and Report" by Jensen and Wirth. The Turbo Pascal compiler has been one of the best-selling series of compilers of all time, and made the language particularly popular on the PC platform, thanks to its balance of simplicity and power.

Turbo Pascal introduced an Integrated Development Environment (IDE) where you could edit the code (in a WordStar compatible editor), run the compiler, see the errors, and jump back to the lines containing those errors. It sounds trivial now, but previously you had to quit the editor, return to DOS; run the command-line compiler, write down the error lines, open the editor and jump there.

Moreover Borland sold Turbo Pascal for 49 dollars, where Microsoft's Pascal compiler was sold for a few hundred. Turbo Pascal's many years of success contributed to Microsoft's eventual cancellation of its Pascal compiler product.

Delphi's Pascal

After 9 versions of Turbo and Borland Pascal compilers, which gradually extended the language, Borland released Delphi in 1995, turning Pascal into a visual programming language.

Delphi extends the Pascal language in a number of ways, including many object-oriented extensions which are different from other flavors of Object Pascal, including those in the Borland Pascal with Objects compiler.

Next Chapter: Coding in Pascal

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 2 Coding in Pascal

Before we move on to the subject of writing Pascal language statements, it is important to highlight a couple of elements of Pascal coding style. The question I'm addressing here is this: Besides the syntax rules, how should you write code? There isn't a single answer to this question, since personal taste can dictate different styles. However, there are some principles you need to know regarding comments, uppercase, spaces, and the so-called pretty-printing. In general, the goal of any coding style is clarity. The style and formatting decisions you make are a form of shorthand, indicating the purpose of a given piece of code. An essential tool for clarity is consistency-whatever style you choose, be sure to follow it throughout a project.

Comments

In Pascal, comments are enclosed in either braces or parentheses followed by a star. Delphi also accepts the C++ style comments, which can span to the end of the line:

```
{this is a comment}
(* this is another comment *)
// this is a comment up to the end of the line
```

The first form is shorter and more commonly used. The second form was often preferred in Europe because many European keyboards lack the brace symbol. The third form of comments has been borrowed from C++ and is available only in the 32-bit versions of Delphi. Comments up to the end of the line are very helpful for short comments and for commenting out a line of code.

> In the listings of the book I'll try to mark comments as italic (and keywords in bold), to be consistent with the default Delphi syntax highlighting.

Having three different forms of comments can be helpful for making nested comments. If you want to comment out several lines of source code to disable them, and these lines contain some real comments, you cannot use the same comment identifier:

```
{  ... code
{comment, creating problems}
... code }
```

With a second comment identifier, you can write the following code, which is correct:

```
{  ... code
//this comment is OK
... code }
```

Note that if the open brace or parenthesis-star is followed by the dollar sign ($), it becomes a compiler directive, as in {$X+}.

> Actually, compiler directives are still comments. For example, {$X+ This is a comment} is legal. It's both a valid directive and a comment, although sane programmers will probably tend to separate directives and comments.

Use of Uppercase

The Pascal compiler (unlike those in other languages) ignores the case (capitalization) of characters. Therefore, the identifiers Myname, MyName, myname, myName, and MYNAME are all exactly equivalent. On the whole, this is definitely a positive, since in case-sensitive languages, many syntax errors are caused by incorrect capitalization.

> Note: There is only one exception to the case-insensitive rule of Pascal: the Register procedure of a components' package must start with the uppercase R, because of a C++Builder compatibility issue.

There are a couple of subtle drawbacks, however. First, you must be aware that these identifiers really are the same, so you must avoid using them as different elements. Second, you should try to be consistent in the use of uppercase letters, to improve the readability of the code.

A consistent use of case isn't enforced by the compiler, but it is a good habit to get into. A common approach is to capitalize only the first letter of each identifier. When an identifier is made up of several consecutive words (you cannot insert a space in an identifier), every first letter of a word should be capitalized:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

Other elements completely ignored by the compiler are the spaces, new lines, and tabs you add to the source code. All these elements are collectively known as white space. White space is used only to improve code readability; it does not affect the compilation.

Unlike BASIC, Pascal allows you to write a statement on several lines of code, splitting a long instruction on two or more lines. The drawback (at least for many BASIC programmers) of allowing statements on more than one line is that you have to remember to add a semicolon to indicate the end of a statement, or more precisely, to separate a statement from the next one. Notice that the only restriction in splitting programming statements on different lines is that a string literal may not span several lines.

Again, there are no fixed rules on the use of spaces and multiple-line statements, just some rules of thumb:

- The Delphi editor has a vertical line you can place after 60 or 70 characters. If you use this line and try to avoid surpassing this limit, your source code will look better when you print it on paper. Otherwise long lines may get broken at any position, even in the middle of a word, when you print them.
- When a function or procedure has several parameters, it is common practice to place the parameters on different lines.
- You can leave a line completely white (blank) before a comment or to divide a long piece of code in smaller portions. Even this simple idea can improve the readability of the code, both on screen and when you print it.
- Use spaces to separate the parameters of a function call, and maybe even a space before the initial open parenthesis. Also keep operands of an expression separated. I know that some programmers will disagree with these ideas, but I insist: Spaces are free; you don't pay for them. (OK, I know that they use up disk space and modem connection time when you upload or download a file, but this is less and less relevant, nowadays.)

Pretty-Printing

The last suggestion on the use of white spaces relates to the typical Pascal language-formatting style, known as pretty-printing. This rule is simple: Each time you need to write a compound statement, indent it two spaces to the right of the rest of the current statement. A compound statement inside another compound statement is indented four spaces, and so on:

```
if ... then
  statement;

if ... then
begin
  statement1;
  statement2;
end;

if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

The above formatting is based on pretty-printing, but programmers have different interpretations of this general rule. Some programmers indent the begin and end statements to the level of the inner code, some of them indent begin and end and then indent the internal code once more, other programmers put the begin in the line of the if condition. This is mostly a matter of personal taste.

A similar indented format is often used for lists of variables or data types, and to continue a statement from the previous line:

```
type
  Letters = set of Char;
var
  Name: string;
begin
   { long comment and long statement, going on in the
     following line and indented two spaces }
   MessageDlg ('This is a message',
     mtInformation, [mbOk], 0);
```

Of course, any such convention is just a suggestion to make the code more readable to other programmers, and it is completely ignored by the compiler. I've tried to use this rule consistently in all of the samples and code fragments in this book. Delphi source code, manuals, and Help examples use a similar formatting style.

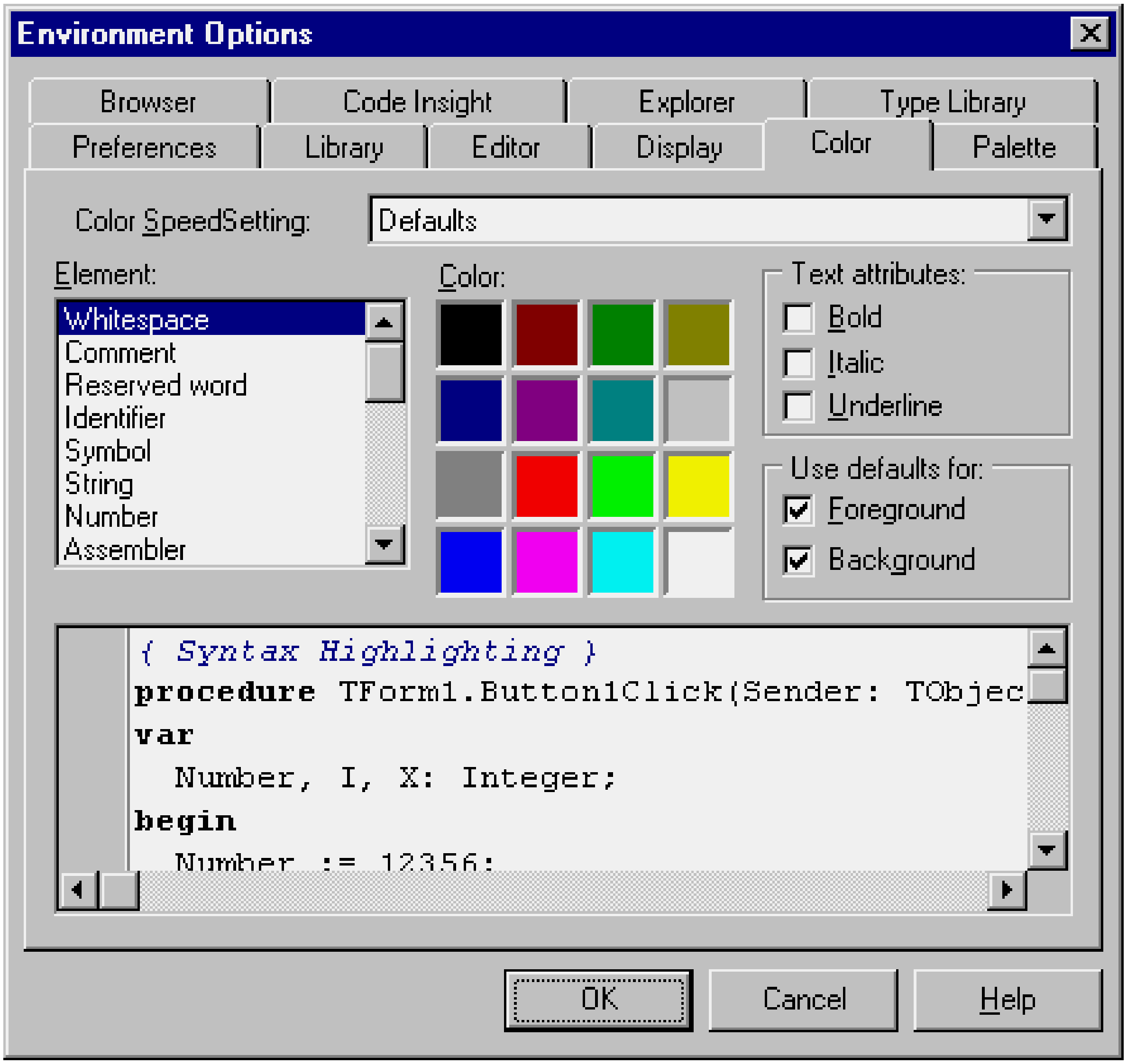
Syntax Highlighting

To make it easier to read and write Pascal code, the Delphi editor has a feature called color syntax highlighting. Depending on the meaning in Pascal of the words you type in the editor, they are displayed using different colors. By default, keywords are in bold, strings and comments are in color (and often in italic), and so on.

Reserved words, comments, and strings are probably the three elements that benefit most from this feature. You can see at a glance a misspelled keyword, a string not properly terminated, and the length of a multiple-line comment.

You can easily customize the syntax highlight settings using the Editor Colors page of the Environment Options dialog box (see Figure 2.1). If you work by yourself, choose the colors you like. If you work closely with other programmers, you should all agree on a standard color scheme. I find that working on a computer with a different syntax coloring than the one I am used to is really difficult.

FIGURE 2.1: The dialog box used to set the color syntax highlighting.



Color SpeedSetting: Defaults

Element:
- Whitespace
- Comment
- Reserved word
- Identifier
- Symbol
- String
- Number
- Assembler

Text attributes:
- □ Bold
- □ Italic
- □ Underline

Use defaults for:
- ☑ Foreground
- ☑ Background

```
{ Syntax Highlighting }
procedure TForm1.Button1Click(Sender: TObjec
var
  Number, I, X: Integer;
begin
  Number := 12356;
```

Note: In this book I've tried to apply a sort of syntax highlighting to the source code listings. I hope this actually makes them more readable.

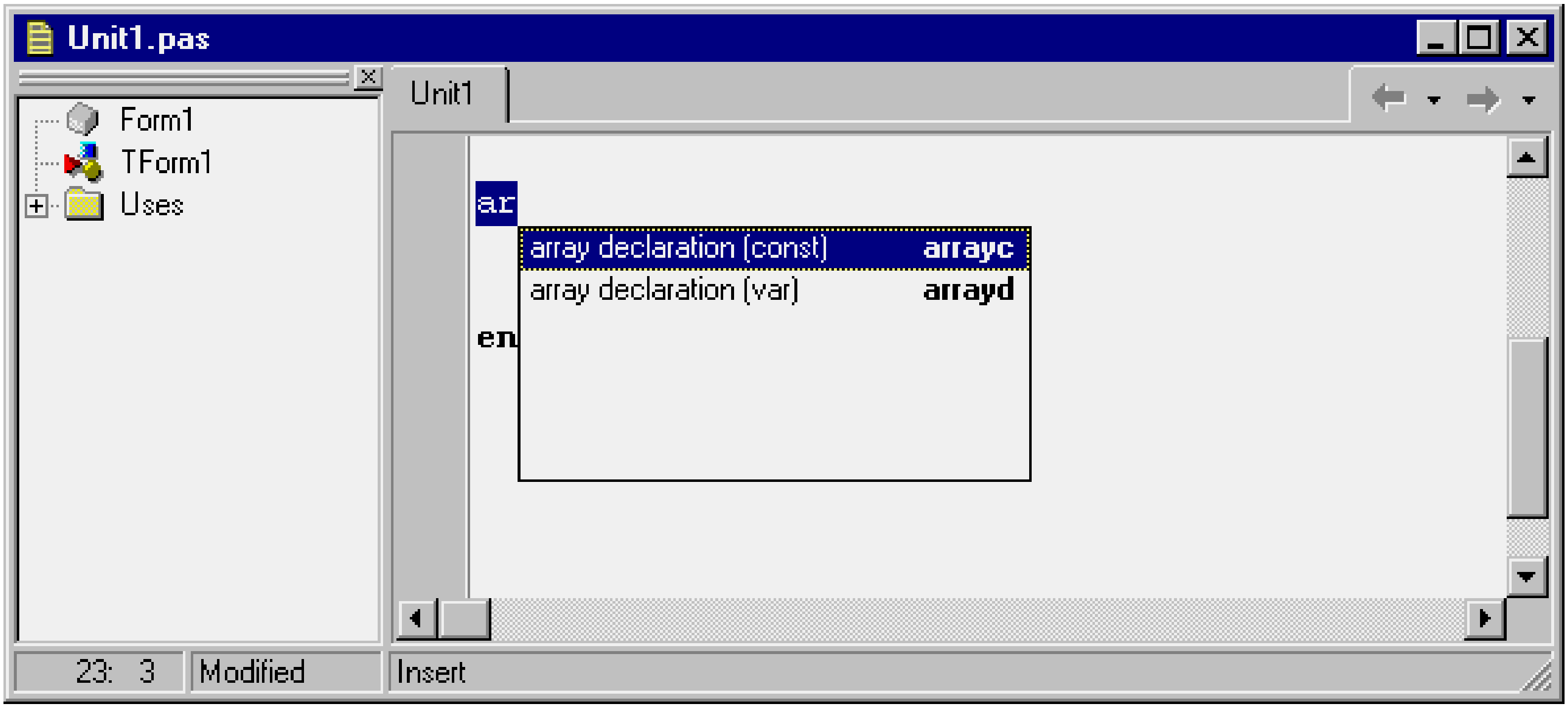
Using Code Templates

Delphi 3 introduced a new feature related to source code editing. Because when writing Pascal language statements you often repeat the same sequence of keywords, Borland has provided a new feature called Code Templates. A code template is simply a piece of code related with a shorthand. You type the shorthand, then press Ctrl+J, and the full piece of code appears. For example, if you type arrayd, and then press Ctrl+J, the Delphi editor will expand your text into:

```
array [0..] of ;
```

Since the predefined code templates usually include several versions of the same construct, the shortcut generally terminates with a letter indicating which of the versions you are interested in. However, you can also type only the initial part of the shortcut. For example, if you type ar and then press Ctrl+J, the editor will display a local menu with a list of the available choices with a short description, as you can see in Figure 2.2.

Figure 2.2: Code Templates selection


Unit1.pas
Form1
TForm1
Uses
Unit1
ar
en
array declaration (const) arrayc
array declaration (var) arrayd
23: 3
Modified
Insert

You can fully customize the code templates by modifying the existing ones or adding your own common code pieces. If you do this, keep in mind that the text of a code template generally includes the '|' character to indicate where the cursor should jump to after the operation, that is, where you start typing to complete the template with custom code.

Language Statements

Once you have defined some identifiers, you can use them in statements and in the expressions that are part of some statements. Pascal offers several statements and expressions. Let's look at keywords, expressions, and operators first.

Keywords

Keywords are all the Object Pascal reserved identifiers, which have a role in the language. Delphi's help distinguishes between reserved words and directives: Reserved words cannot be used as identifiers, while directives should not be used as such, even if the compiler will accept them. In practice, you should not use any keywords as an identifier.

In Table 2.1 you can see a complete list of the identifiers having a specific role in the Object Pascal language (in Delphi 4), including keywords and other reserved words.

Table 2.1: Keywords and other reserved words in the Object Pascal language

Keyword	Role
absolute	directive (variables)
abstract	directive (method)

and	operator (boolean)
array	type
as	operator (RTTI)
asm	statement
assembler	backward compatibility (asm)
at	statement (exceptions)
automated	access specifier (class)
begin	block marker
case	statement
cdecl	function calling convention
class	type
const	declaration or directive (parameters)
constructor	special method
contains	operator (set)
default	directive (property)
destructor	special method
dispid	dispinterface specifier
dispinterface	type
div	operator
do	statement
downto	statement (for)
dynamic	directive (method)
else	statement (if or case)
end	block marker
except	statement (exceptions)
export	backward compatibility (class)
exports	declaration
external	directive (functions)
far	backward compatibility (class)
file	type
finalization	unit structure
finally	statement (exceptions)
for	statement
forward	function directive
function	declaration
goto	statement
if	statement
implementation	unit structure
implements	directive (property)
in	operator (set) - project structure
index	directive (dipinterface)
inherited	statement
initialization	unit structure
inline	backward compatibility (see asm)
interface	type

is	operator (RTTI)
label	declaration
library	program structure
message	directive (method)
mod	operator (math)
name	directive (function)
near	backward compatibility (class)
nil	value
nodefault	directive (property)
not	operator (boolean)
object	backward compatibility (class)
of	statement (case)
on	statement (exceptions)
or	operator (boolean)
out	directive (parameters)
overload	function directive
override	function directive
package	program structure (package)
packed	directive (record)
pascal	function calling convention
private	access specifier (class)
procedure	declaration
program	program structure
property	declaration
protected	access specifier (class)
public	access specifier (class)
published	access specifier (class)
raise	statement (exceptions)
read	property specifier
readonly	dispatch interface specifier
record	type
register	function calling convention
reintroduce	function directive
repeat	statement
requires	program structure (package)
resident	directive (functions)
resourcestring	type
safecall	function calling convention
set	type
shl	operator (math)
shr	operator (math)
stdcall	function calling convention
stored	directive (property)
string	type
then	statement (if)

threadvar	declaration
to	statement (for)
try	statement (exceptions)
type	declaration
unit	unit structure
until	statement
uses	unit structure
var	declaration
virtual	directive (method)
while	statement
with	statement
write	property specifier
writeonly	dispatch interface specifier
xor	operator (boolean)

Expressions and Operators

There isn't a general rule for building expressions, since they mainly depend on the operators being used, and Pascal has a number of operators. There are logical, arithmetic, Boolean, relational, and set operators, plus some others. Expressions can be used to determine the value to assign to a variable, to compute the parameter of a function or procedure, or to test for a condition. Expressions can include function calls, too. Every time you are performing an operation on the value of an identifier, rather than using an identifier by itself, that is an expression.

Expressions are common to most programming languages. An expression is any valid combination of constants, variables, literal values, operators, and function results. Expressions can also be passed to value parameters of procedures and functions, but not always to reference parameters (which require a value you can assign to).

Operators and Precedence

If you have ever written a program in your life, you already know what an expression is. Here, I'll highlight specific elements of Pascal operators. You can see a list of the operators of the language, grouped by precedence, in Table 2.1.

> Contrary to most other programming languages, the and and or operators have precedence compared to the relational one. So if you write a < b and c < d, the compiler will try to do the and operation first, resulting in a compiler error. For this reason you should enclose each of the < expression in parentheses: (a < b) and (c < d).

Some of the common operators have different meanings with different data types. For example, the + operator can be used to add two numbers, concatenate two strings, make the union of two sets, and even add an offset to a PChar pointer. However, you cannot add two characters, as is possible in C.

Another strange operator is div. In Pascal, you can divide any two numbers (real or integers) with the / operator, and you'll invariably get a real-number result. If you need to divide two integers and want an integer result, use the div operator instead.

Table 2.2: Pascal Language Operators, Grouped by Precedence

Unary Operators (Highest Precedence)

@	Address of the variable or function (returns a pointer)
not	Boolean or bitwise not
Multiplicative and Bitwise Operators	
*	Arithmetic multiplication or set intersection
/	Floating-point division
div	Integer division
mod	Modulus (the remainder of integer division)
as	Allows a type-checked type conversion among at runtime (part of the RTTI support)
and	Boolean or bitwise and
shl	Bitwise left shift
shr	Bitwise right shift
Additive Operators	
+	Arithmetic addition, set union, string concatenation, pointer offset addition
-	Arithmetic subtraction, set difference, pointer offset subtraction
or	Boolean or bitwise or
xor	Boolean or bitwise exclusive or
Relational and Comparison Operators (Lowest Precedence)	
=	Test whether equal
<>	Test whether not equal
<	Test whether less than
>	Test whether greater than
<=	Test whether less than or equal to, or a subset of a set
>=	Test whether greater than or equal to, or a superset of a set
in	Test whether the item is a member of the set
is	Test whether object is type-compatible (another RTTI operator)

Set Operators

The set operators include union (+), difference (-), intersection (*),membership test (in), plus some relational operators. To add an element to a set, you can make the union of the set with another one that has only the element you need. Here's a Delphi example related to font styles:

```
Style := Style + [fsBold];
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

As an alternative, you can use the standard Include and Exclude procedures, which are much more efficient (but cannot be used with component properties of the set type, because they require an l-value parameter):

```
Include (Style, fsBold);
```

Conclusion

Now that we know the basic layout of a Pascal program we are ready to start understanding its meaning in detail. We'll start by exploring the definition of predefined and user defined data types, then we'll move along to the use of the keywords to form programming statements.

Next Chapter: Types, Variables, and Constants

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 3 Types, Variables, and Constants

The original Pascal language was based on some simple notions, which have now become quite common in programming languages. The first is the notion of data type. The type determines the values a variable can have, and the operations that can be performed on it. The concept of type is stronger in Pascal than in C, where the arithmetic data types are almost interchangeable, and much stronger than in the original versions of BASIC, which had no similar concept.

Variables

Pascal requires all variables to be declared before they are used. Every time you declare a variable, you must specify a data type. Here are some sample variable declarations:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

The var keyword can be used in several places in the code, such as at the beginning of the code of a function or procedure, to declare variables local to the routine, or inside a unit to declare global variables. After the var keyword comes a list of variable names, followed by a colon and the name of the data type. You can write more than one variable name on a single line, as in the last statement above.

Once you have defined a variable of a given type, you can perform on it only the operations supported by its data type. For example, you can use the Boolean value in a test and the integer value in a numerical expression. You cannot mix Booleans and integers (as you can with the C language).

Using simple assignments, we can write the following code:

```
Value := 10;
IsCorrect := True;
```

But the next statement is not correct, because the two variables have different data types:

```
Value := IsCorrect; // error
```

If you try to compile this code, Delphi issues a compiler error with this description: Incompatible types: 'Integer' and 'Boolean'. Usually, errors like this are programming errors, because it does not make sense to assign a True or False value to a variable of the Integer data type. You should not blame Delphi for these errors. It only warns you that there is something wrong in the code.

Of course, it is often possible to convert the value of a variable from one type into a different type. In some cases, this conversion is automatic, but usually you need to call a specific system function that changes the internal representation of the data.

In Delphi you can assign an initial value to a global variable while you declare it. For example, you can write:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

This initialization technique works only for global variables, not for variables declared inside the scope of a procedure or method.

Constants

Pascal also allows the declaration of constants to name values that do not change during program execution. To declare a constant you don't need to specify a data type, but only assign an initial value. The compiler will look at the value and automatically use its proper data type. Here are some sample declarations:

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantù';
```

Delphi determines the constant's data type based on its value. In the example above, the Thousand constant is assumed to be of type SmallInt, the smallest integral type which can hold it. If you want to tell Delphi to use a specific type you can simply add the type name in the declaration, as in:

```
const
  Thousand: Integer = 1000;
```

When you declare a constant, the compiler can choose whether to assign a memory location to the constant, and save its value there, or to duplicate the actual value each time the constant is used. This second approach makes sense particularly for simple constants.

> Note: The 16-bit version of Delphi allows you to change the value of a typed constant at run-time, as if it was a variable. The 32-bit version still permits this behavior for backward compatibility when you enable the $J compiler directive, or use the corresponding Assignable typed constants check box of the Compiler page of the Project Options dialog box. Although this is the default, you are strongly advised not to use this trick as a general programming technique. Assigning a new value to a constant disables all the compiler optimizations on constants. In such a case, simply declare a variable, instead.

Resource String Constants

When you define a string constant, instead of writing:

```
const
  AuthorName = 'Marco Cantù';
```

starting with Delphi 3 you can write the following:

```
resourcestring
  AuthorName = 'Marco Cantù';
```

In both cases you are defining a constant; that is, a value you don't change during program execution. The difference is only in the implementation. A string constant defined with the resourcestring directive is stored in the resources of the program, in a string table.

To see this capability in action, you can look at the ResStr example, which has a button with the following code:

```
resourcestring
  AuthorName = 'Marco Cantù';
  BookName = 'Essential Pascal';

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage (BookName + #13 + AuthorName);
end;
```

The output of the two strings appears on separate lines because the strings are separated by the newline character (indicated by its numerical value in the #13 character-type constant).

The interesting aspect of this program is that if you examine it with a resource explorer (there is one available among the examples that ship with Delphi) you'll see the new strings in the resources. This means that the strings are not part of the compiled code but stored in a separate area of the executable file (the EXE file).

> Note: In short, the advantage of resources is in an efficient memory handling performed by Windows and in the possibility of localizing a program (translating the strings to a different language) without having to modify its source code.

Data Types

In Pascal there are several predefined data types, which can be divided into three groups: ordinal types, real types, and strings. We'll discuss ordinal and real types in the following sections, while strings are covered later in this chapter. In this section I'll also introduce some types defined by the Delphi libraries (not predefined by the compiler), which can be considered predefined types.

Delphi also includes a non-typed data type, called variant, and discussed in Chapter 10 of this book. Strangely enough a variant is a type without proper type-checking. It was introduced in Delphi 2 to handle OLE Automation.

Ordinal Types

Ordinal types are based on the concept of order or sequence. Not only can you compare two values to see which is higher, but you can also ask for the value following or preceding a given value or compute the lowest or highest possible value.

The three most important predefined ordinal types are Integer, Boolean, and Char (character). However, there are a number of other related types that have the same meaning but a different internal representation and range of values. The following Table 3.1 lists the ordinal data types used for representing numbers.

Table 3.1: Ordinal data types for numbers

Size	Signed Range	Unsigned Range
8 bits	ShortInt -128 to 127	Byte 0 to 255
16 bits	SmallInt -32768 to 32767	Word 0 to 65,535
32 bits	LongInt -2,147,483,648 to 2,147,483,647	LongWord (since Delphi 4) 0 to 4,294,967,295
64 bits	Int64	
16/32 bits	Integer	Cardinal

As you can see, these types correspond to different representations of numbers, depending on the number of bits used to express the value, and the presence or absence of a sign bit. Signed values can be positive or negative, but have a smaller range of values, because one less bit is available for the value itself. You can refer to the Range example, discussed in the next section, for the actual range of values of each type.

The last group (marked as 16/32) indicates values having a different representation in the 16-bit and 32-bit versions of Delphi. Integer and Cardinal are frequently used, because they correspond to the native representation of numbers in the CPU.

Integral Types in Delphi 4

In Delphi 3, the 32-bit unsigned numbers indicated by the Cardinal type were actually 31-bit values, with a range up to 2 gigabytes. Delphi 4 introduced a new unsigned numeric type, LongWord, which uses a truly 32-bit value up to 4 gigabytes. The Cardinal type is now an alias of the new LongWord type. LongWord permits 2GB more data to be addressed by an unsigned number, as mentioned above. Moreover, it corresponds to the native representation of numbers in the CPU.

Another new type introduced in Delphi 4 is the Int64 type, which represents integer numbers with up to 18 digits. This new type is fully supported by some of the ordinal type routines (such as High and Low), numeric routines (such as Inc and Dec), and string-conversion routines (such as IntToStr). For the opposite conversion, from a string to a number, there are two new specific functions: StrToInt64 and StrToInt64Def.

Boolean

Boolean values other than the Boolean type are seldom used. Some Boolean values with specific representations are required by Windows API functions. The types are ByteBool, WordBool, and LongBool.

In Delphi 3 for compatibility with Visual Basic and OLE automation, the data types ByteBool, WordBool, and LongBool were modified to represent the value True with -1, while the value False is still 0. The Boolean data type remains unchanged (True is 1, False is 0). If you've used explicit typecasts in your Delphi 2 code, porting the code to later versions of Delphi might result in errors.

Characters

Finally there are two different representation for characters: ANSIChar and WideChar. The first type represents 8-bit characters, corresponding to the ANSI character set traditionally used by Windows; the second represents 16-bit characters, corresponding to the new Unicode characters supported by Windows NT, and only partially by Windows 95 and 98. Most of the time you'll simply use the Char type, which in Delphi 3 corresponds to ANSIChar. Keep in mind, anyway, that the first 256 Unicode characters correspond exactly to the ANSI characters.

Constant characters can be represented with their symbolic notation, as in 'k', or with a numeric notation, as in #78. The latter can also be expressed using the Chr function, as in Chr (78). The opposite conversion can be done with the Ord function.

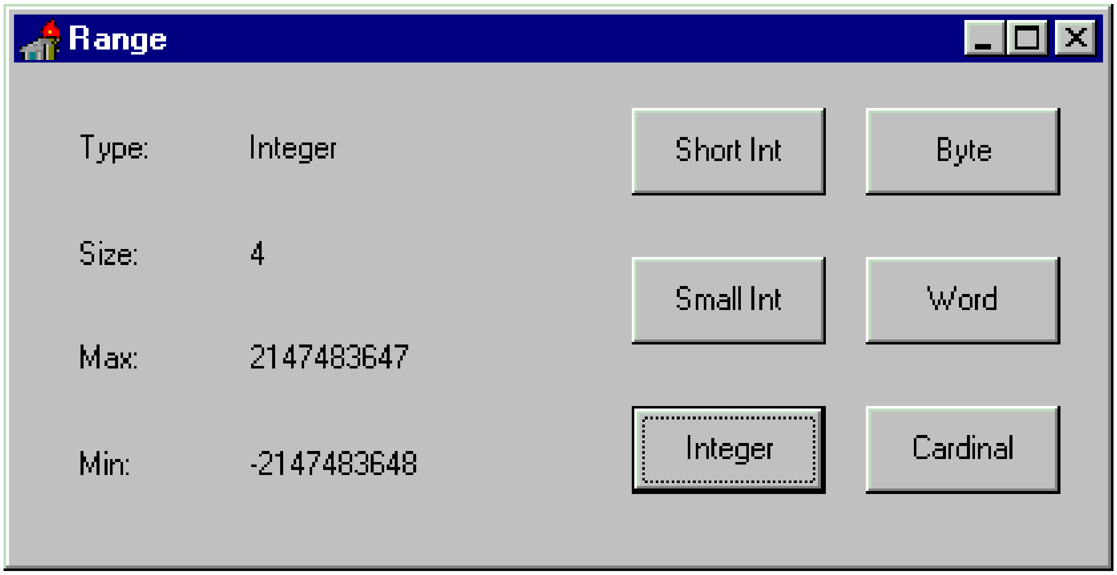
It is generally better to use the symbolic notation when indicating letters, digits, or symbols. When referring to special characters, instead, you'll generally use the numeric notation. The following list includes some of the most commonly used special characters:

- #9 tabulator
- #10 newline
- #13 carriage return (enter key)

The Range Example

To give you an idea of the different ranges of some of the ordinal types, I've written a simple Delphi program named Range. Some results are shown in Figure 3.1.

FIGURE 3.1: The Range example displays some information about ordinal data types (Integers in this case).



Type: Integer

Size: 4

Max: 2147483647

Min: -2147483648

The Range program is based on a simple form, which has six buttons (each named after an ordinal data type) and some labels for categories of information, as you can see in Figure 3.1. Some of the labels are used to hold static text, others to show the information about the type each time one of the buttons is pressed.

Every time you press one of the buttons on the right, the program updates the labels with the output. Different labels show the data type, number of bytes used, and the maximum and minimum values the data type can store. Each button has its own OnClick event-response method because the code used to compute the three values is slightly different from button to button. For example, here is the source code of the OnClick event for the Integer button (BtnInteger):

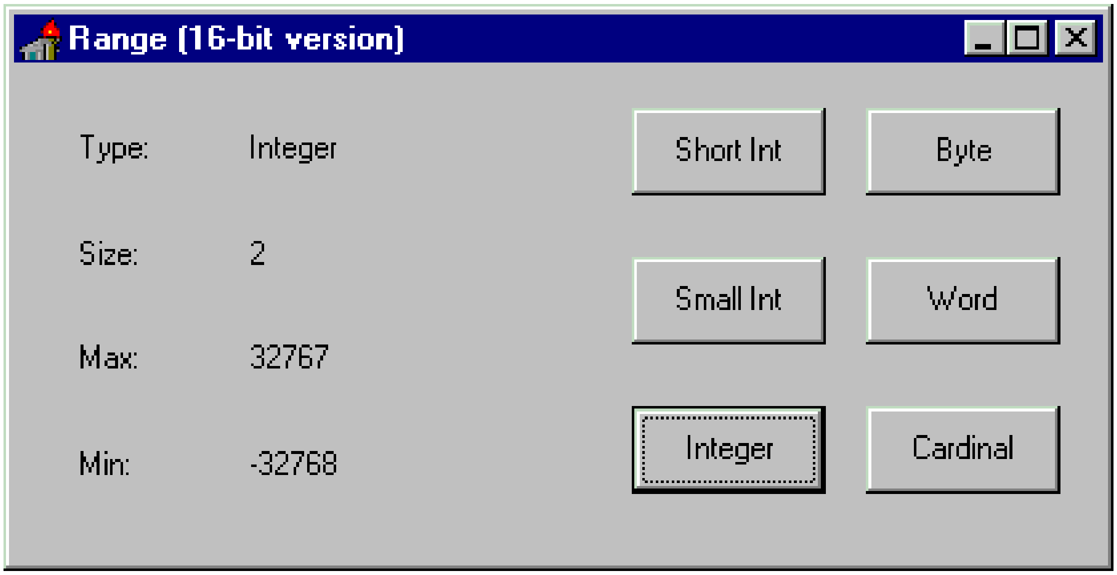
```
procedure TFormRange.BtnIntegerClick(Sender: TObject);
begin
  LabelType.Caption := 'Integer';
  LabelSize.Caption := IntToStr (SizeOf (Integer));
  LabelMax.Caption := IntToStr (High (Integer));
  LabelMin.Caption := IntToStr (Low (Integer));
end;
```

If you have some experience with Delphi programming, you can examine the source code of the program to understand how it works.

For beginners, it's enough to note the use of three functions: SizeOf, High, and Low. The results of the last two functions are ordinals of the same kind (in this case, integers), and the result of the SizeOf function is always an integer. The return value of each of these functions is first translated into strings using the IntToStr function, then copied to the captions of the three labels.

The methods associated with the other buttons are very similar to the one above. The only real difference is in the data type passed as a parameter to the various functions. Figure 3.2 shows the result of executing this same program under Windows 95 after it has been recompiled with the 16-bit version of Delphi. Comparing Figure 3.1 with Figure 3.2, you can see the difference between the 16-bit and 32-bit Integer data types.

FIGURE 3.2: The output of the 16-bit version of the Range example, again showing information about integers.



Range (16-bit version)

Type: Integer

Size: 2

Max: 32767

Min: -32768

The size of the Integer type varies depending on the CPU and operating system you are using. In 16-bit Windows, an Integer variable is two bytes wide. In 32-bit Windows, an Integer is four bytes wide. For this reason, when you recompile the Range example, you get a different output.

The two different representations of the Integer type are not a problem, as long as your program doesn't make any assumptions about the size of integers. If you happen to save an Integer to a file using one version and retrieve it with another, though, you're going to have some trouble. In this situation, you should choose a platform-independent data type (such as LongInt or SmallInt). For mathematical computation or generic code, your best bet is to stick with the standard integral representation for the specific platform--that is, use the Integer type--because this is what the CPU likes best. The Integer type should be your first choice when handling integer numbers. Use a different representation only when there is a compelling reason to do so.

Ordinal Types Routines

There are some system routines (routines defined in the Pascal language and in the Delphi system unit) that work on ordinal types. They are shown in Table 3.2. C++ programmers should notice that the two versions of the Inc procedure, with one or two parameters, correspond to the ++ and += operators (the same holds for the Dec procedure).

Table 3.2: System Routines for Ordinal Types

Routine	Purpose
Dec	Decrements the variable passed as parameter, by one or by the value of the optional second parameter.
Inc	Increments the variable passed as parameter, by one or by the specified value.

Odd	Returns True if the argument is an odd number.
Pred	Returns the value before the argument in the order determined by the data type, the predecessor.
Succ	Returns the value after the argument, the successor.
Ord	Returns a number indicating the order of the argument within the set of values of the data type.
Low	Returns the lowest value in the range of the ordinal type passed as its parameter.
High	Returns the highest value in the range of the ordinal data type.

Notice that some of these routines, when applied to constants, are automatically evaluated by the compiler and replaced by their value. For example if you call High(X) where X is defined as an Integer, the compiler can simply replace the expression with the highest possible value of the Integer data type.

Real Types

Real types represent floating-point numbers in various formats. The smallest storage size is given by Single numbers, which are implemented with a 4-byte value. Then there are Double floating-point numbers, implemented with 8 bytes, and Extended numbers, implemented with 10 bytes. These are all floating-point data types with different precision, which correspond to the IEEE standard floating-point representations, and are directly supported by the CPU numeric coprocessor, for maximum speed.

In Delphi 2 and Delphi 3 the Real type had the same definition as in the 16-bit version; it was a 48-bit type. But its usage was deprecated by Borland, which suggested that you use the Single, Double, and Extended types instead. The reason for their suggestion is that the old 6-byte format is neither supported by the Intel CPU nor listed among the official IEEE real types. To completely overcome the problem, Delphi 4 modifies the definition of the Real type to represent a standard 8-byte (64-bit) floating-point number.

In addition to the advantage of using a standard definition, this change allows components to publish properties based on the Real type, something Delphi 3 did not allow. Among the disadvantages there might be compatibility problems. If necessary, you can overcome the possibility of incompatibility by sticking to the Delphi 2 and 3 definition of the type; do this by using the following compiler option:

```
{$REALCOMPATIBILITY ON}
```

There are also two strange data types: Comp describes very big integers using 8 bytes (which can hold numbers with 18 decimal digits); and Currency (not available in 16-bit Delphi) indicates a fixed-point decimal value with four decimal digits, and the same 64-bit representation as the Comp type. As the name implies, the Currency data type has been added to handle very precise monetary values, with four decimal places.

We cannot build a program similar to the Range example with real data types, because we cannot use the High and Low functions or the Ord function on real-type variables. Real types represent (in theory) an infinite set of numbers; ordinal types represent a fixed set of values.

> Note: Let me explain this better. when you have the integer 23 you can determine which is the following value. Integers are finite (they have a determined range and they have an order). Floating point numbers are infinite even within a small range, and have no order: in fact, how many values are there between 23 and 24? And which number follows 23.46? It is 23.47, 23.461, or 23.4601? That's really hard to know!

For this reason, it makes sense to ask for the ordinal position of the character w in the range of the Char data type, but it makes no sense at all to ask the same question about 7143.1562 in the range of a floating-point data type. Although you can indeed know whether one real number has a higher value than another, it makes no sense to ask how many real numbers exist before a given number (this is the meaning of the Ord function).

Real types have a limited role in the user interface portion of the code (the Windows side), but they are fully supported by Delphi, including the database side. The support of IEEE standard floating-point types makes the Object Pascal language completely appropriate for the wide range of programs that require numerical computations. If you are interested in this aspect, you can look at the arithmetic functions provided by Delphi in the system unit (see the Delphi Help for more details).

Note: Delphi also has a Math unit that defines advanced mathematical routines, covering trigonometric functions (such as the ArcCosh function), finance (such as the InterestPayment function), and statistics (such as the MeanAndStdDev procedure). There are a number of these routines, some of which sound quite strange to me, such as the MomentSkewKurtosis procedure (I'll let you find out what this is).

Date and Time

Delphi uses real types also to handle date and time information. To be more precise Delphi defines a specific TDateTime data type. This is a floating-point type, because the type must be wide enough to store years, months, days, hours, minutes, and seconds, down to millisecond resolution in a single variable. Dates are stored as the number of days since 1899-12-30 (with negative values indicating dates before 1899) in the integer part of the TDateTime value. Times are stored as fractions of a day in the decimal part of the value.

TDateTime is not a predefined type the compiler understands, but it is defined in the system unit as:

```
type
  TDateTime = type Double;
```

Using the TDateTime type is quite easy, because Delphi includes a number of functions that operate on this type. You can find a list of these functions in Table 3.3.

Table 3.3: System Routines for the TDateTime Type

Routine	Description
Now	Returns the current date and time into a single TDateTime value.
Date	Returns only the current date.
Time	Returns only the current time.
DateTimeToStr	Converts a date and time value into a string, using default formatting; to have more control on the conversion use the FormatDateTime function instead.
DateTimeToString	Copies the date and time values into a string buffer, with default formatting.
DateToStr	Converts the date portion of a TDateTime value into a string.
TimeToStr	Converts the time portion of a TDateTime value into a string.
FormatDateTime	Formats a date and time using the specified format; you can specify which values you want to see and which format to use, providing a complex format string.
StrToDateTime	Converts a string with date and time information to a TDateTime value, raising an exception in case of an error in the format of the string.
StrToDate	Converts a string with a date value into the TDateTime format.
StrToTime	Converts a string with a time value into the TDateTime format.
DayOfWeek	Returns the number corresponding to the day of the week of the TDateTime value passed as parameter.

DecodeDate	Retrieves the year, month, and day values from a date value.
DecodeTime	Retrieves out of a time value.
EncodeDate	Turns year, month, and day values into a TDateTime value.
EncodeTime	Turns hour, minute, second, and millisecond values into a TDateTime value.

To show you how to use this data type and some of its related routines, I've built a simple example, named TimeNow. The main form of this example has a Button and a ListBox component. When the program starts it automatically computes and displays the current time and date. Every time the button is pressed, the program shows the time elapsed since the program started.

Here is the code related to the OnCreate event of the form:

```
procedure TFormTimeNow.FormCreate(Sender: TObject);
begin
  StartTime := Now;
  ListBox1.Items.Add (TimeToStr (StartTime));
  ListBox1.Items.Add (DateToStr (StartTime));
  ListBox1.Items.Add ('Press button for elapsed time');
end;
```

The first statement is a call to the Now function, which returns the current date and time. This value is stored in the StartTime variable, declared as a global variable as follows:

```
var
  FormTimeNow: TFormTimeNow;
  StartTime: TDateTime;
```

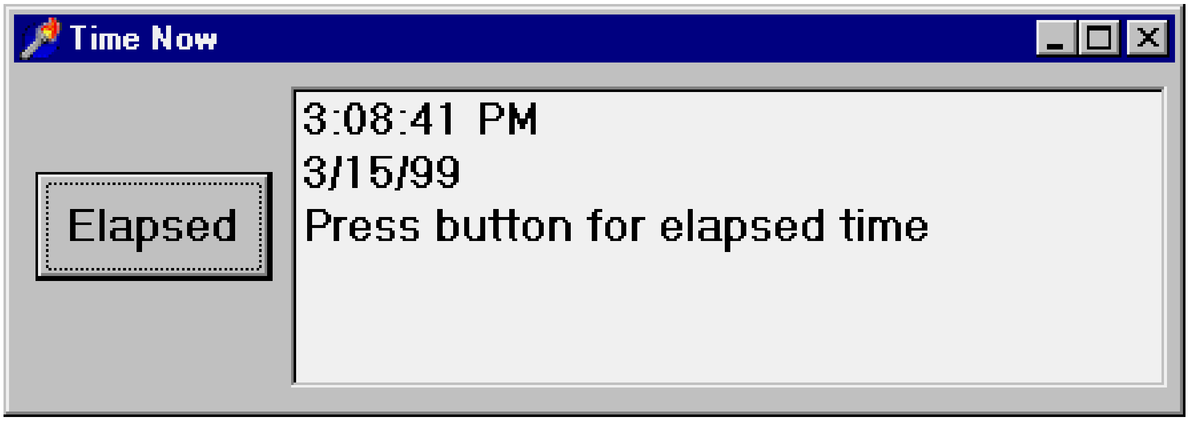
I've added only the second declaration, since the first is provided by Delphi. By default, it is the following:

```
var
  Form1: TForm1;
```

Changing the name of the form, this declaration is automatically updated. Using global variables is actually not the best approach: It should be better to use a private field of the form class, a topic related to object-oriented programming and discussed in Mastering Delphi 4.

The next three statements add three items to the ListBox component on the left of the form, with the result you can see in Figure 3.3. The first line contains the time portion of the TDateTime value converted into a string, the second the date portion of the same value. At the end the code adds a simple reminder.

FIGURE 3.3: The output of the TimeNow example at startup.



3:08:41 PM
3/15/99
Press button for elapsed time

This third string is replaced by the program when the user clicks on the Elapsed button:

```
procedure TFormTimeNow.ButtonElapsedClick(Sender: TObject);
var
  StopTime: TDateTime;
begin
  StopTime := Now;
  ListBox1.Items [2] :=  FormatDateTime ('hh:nn:ss',
    StopTime - StartTime);
end;
```

This code retrieves the new time and computes the difference from the time value stored when the program started. Because we need to use a value that we computed in a different event handler, we had to store it in a global variable. There are actually better alternatives, based on classes.

> **Note:** The code that replaces the current value of the third string uses the index 2. The reason is that the items of a list box are zero-based: the first item is number 0, the second number 1, and the third number 2. More on this as we cover arrays.

Besides calling TimeToStr and DateToStr you can use the more powerful FormatDateTime function, as I've done in the last method above (see the Delphi Help file for details on the formatting parameters). Notice also that time and date values are transformed into strings depending on Windows international settings. Delphi reads these values from the system, and copies them to a number of global constants declared in the SysUtils unit. Some of them are:

```
DateSeparator: Char;
ShortDateFormat: string;
LongDateFormat: string;
TimeSeparator: Char;
TimeAMString: string;
TimePMString: string;
ShortTimeFormat: string;
LongTimeFormat: string;
ShortMonthNames: array [1..12] of string;
LongMonthNames: array [1..12] of string;
ShortDayNames: array [1..7] of string;
LongDayNames: array [1..7] of string;
```

More global constants relate to currency and floating-point number formatting. You can find the complete list in the Delphi Help file under the topic Currency and date/time formatting variables.

> **Note:** Delphi includes a DateTimePicker component, which provides a sophisticated way to input a date, selecting it from a

calendar.

Specific Windows Types

The predefined data types we have seen so far are part of the Pascal language. Delphi also includes other data types defined by Windows. These data types are not an integral part of the language, but they are part of the Windows libraries. Windows types include new default types (such as DWORD or UINT), many records (or structures), several pointer types, and so on.

Among Windows data types, the most important type is represented by handles, discussed in Chapter 9.

Typecasting and Type Conversions

As we have seen, you cannot assign a variable to another one of a different type. In case you need to do this, there are two choices. The first choice is typecasting, which uses a simple functional notation, with the name of the destination data type:

```
var
  N: Integer;
  C: Char;
  B: Boolean;
begin
  N := Integer ('X');
  C := Char (N);
  B := Boolean (0);
```

You can typecast between data types having the same size. It is usually safe to typecast between ordinal types, or between real types, but you can also typecast between pointer types (and also objects) as long as you know what you are doing.

Casting, however, is generally a dangerous programming practice, because it allows you to access a value as if it represented something else. Since the internal representations of data types generally do not match, you risk hard-to-track errors. For this reason, you should generally avoid typecasting.

The second choice is to use a type-conversion routine. The routines for the various types of conversions are summarized in Table 3.4. Some of these routines work on the data types that we'll discuss in the following sections. Notice that the table doesn't include routines for special types (such as TDateTime or variant) or routines specifically intended for formatting, like the powerful Format and FormatFloat routines.

Table 3.4: System Routines for Type Conversion

Routine	Description
Chr	Converts an ordinal number into an ANSI character.
Ord	Converts an ordinal-type value into the number indicating its order.
Round	Converts a real-type value into an Integer-type value, rounding its value.
Trunc	Converts a real-type value into an Integer-type value, truncating its value.
Int	Returns the Integer part of the floating-point value argument.
IntToStr	Converts a number into a string.
IntToHex	Converts a number into a string with its hexadecimal representation.

StrToInt	Converts a string into a number, raising an exception if the string does not represent a valid integer.
StrToIntDef	Converts a string into a number, using a default value if the string is not correct.
Val	Converts a string into a number (traditional Turbo Pascal routine, available for compatibility).
Str	Converts a number into a string, using formatting parameters (traditional Turbo Pascal routine, available for compatibility).
StrPas	Converts a null-terminated string into a Pascal-style string. This conversion is automatically done for AnsiStrings in 32-bit Delphi. (See the section on strings later in this chapter.)
StrPCopy	Copies a Pascal-style string into a null-terminated string. This conversion is done with a simple PChar cast in 32-bit Delphi. (See the section on strings later in this chapter.)
StrPLCopy	Copies a portion of a Pascal-style string into a null-terminated string.
FloatToDecimal	Converts a floating-point value to record including its decimal representation (exponent, digits, sign).
FloatToStr	Converts the floating-point value to its string representation using default formatting.
FloatToStrF	Converts the floating-point value to its string representation using the specified formatting.
FloatToText	Copies the floating-point value to a string buffer, using the specified formatting.
FloatToTextFmt	As the previous routine, copies the floating-point value to a string buffer, using the specified formatting.
StrToFloat	Converts the given Pascal string to a floating-point value.
TextToFloat	Converts the given null-terminated string to a floating-point value.

NEW Note: In recent versions of Delphi's Pascal compiler, the Round function is based on the FPU processor of the CPU. This processor adopts the so-called "Banker's Rounding", which rounds middle values (as 5.5 or 6.5) up and down depending whether they follow an odd or an even number.

Conclusion

In this chapter we've explored the basic notion of type in Pascal. But the language has another very important feature: It allows programmers to define new custom data types, called user-defined data types. This is the topic of the next chapter.

Next Chapter: User-Defined Data Types

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 4 User-Defined Data Types

Along with the notion of type, one of the great ideas introduced by the Pascal language is the ability to define new data types in a program. Programmers can define their own data types by means of type constructors, such as subrange types, array types, record types, enumerated types, pointer types, and set types. The most important user-defined data type is the class, which is part of the object-oriented extensions of Object Pascal, not covered in this book.

If you think that type constructors are common in many programming languages, you are right, but Pascal was the first language to introduce the idea in a formal and very precise way. There are still few languages with so many mechanisms to define new types.

Named and Unnamed Types

These types can be given a name for later use or applied to a variable directly. When you give a name to a type, you must provide a specific section in the code, such as the following:

```
type
  // subrange definition
  Uppercase = 'A'..'Z';

  // array definition
  Temperatures = array [1..24] of Integer;

  // record definition
  Date = record
    Month: Byte;
    Day: Byte;
    Year: Integer;
  end;

  // enumerated type definition
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);

  // set definition
  Letters = set of Char;
```

Similar type-definition constructs can be used directly to define a variable without an explicit type name, as in the following code:

```
var
  DecemberTemperature: array [1..31] of Byte;
  ColorCode: array [Red..Violet] of Word;
  Palette: set of Colors;
```

Note: In general, you should avoid using unnamed types as in the code above, because you cannot pass them as parameters to routines or declare other variables of the same type. The type compatibility rules of Pascal, in fact, are based on type names, not on the actual definition of the types. Two variables of two identical types are still not compatible, unless their types have exactly the same name, and unnamed types are given internal names by the compiler. Get used to defining a data type each time you need a variable with a complicated structure, and you won't regret the time you've spent in it.

But what do these type definitions mean? I'll provide some descriptions for those who are not familiar with Pascal type constructs. I'll also try to underline the differences from the same constructs in other programming languages, so you might be interested in reading the following sections even if you are familiar with kind of type definitions exemplified above. Finally, I'll show some Delphi examples and introduce some tools that will allow you to access type information dynamically.

Subrange Types

A subrange type defines a range of values within the range of another type (hence the name subrange). You can define a subrange of the Integer type, from 1 to 10 or from 100 to 1000, or you can define a subrange of the Char type, as in:

```
type
  Ten = 1..10;
  OverHundred = 100..1000;
  Uppercase = 'A'..'Z';
```

In the definition of a subrange, you don't need to specify the name of the base type. You just need to supply two constants of that type. The original type must be an ordinal type, and the resulting type will be another ordinal type.

When you have defined a subrange, you can legally assign it a value within that range. This code is valid:

```
var
  UppLetter: UpperCase;
begin
  UppLetter := 'F';
```

But this one is not:

```
var
  UppLetter: UpperCase;
begin
  UppLetter := 'e'; // compile-time error
```

Writing the code above results in a compile-time error, "Constant expression violates subrange bounds." If you write the following code instead:

```
var
  UppLetter: Uppercase;
  Letter: Char;
begin
  Letter :='e';
  UppLetter := Letter;
```

Delphi will compile it. At run-time, if you have enabled the Range Checking compiler option (in the Compiler page of the Project Options dialog box), you'll get a Range check error message.

Note: I suggest that you turn on this compiler option while you are developing a program, so it'll be more robust and easier to debug, as in case of errors you'll get an explicit message and not an undetermined behavior. You can eventually disable the option for the final build of the program, to make it a little faster. However, the difference is really small, and for this reason I suggest you to leave all these run-time checks turned on, even in a shipping program. The same holds true for other run-time checking options, such as overflow and stack checking.

Enumerated Types

Enumerated types constitute another user-defined ordinal type. Instead of indicating a range of an existing type, in an enumeration you list all of the possible values for the type. In other words, an enumeration is a list of values. Here are some examples:

```
type
  Colors = (Red, Yellow, Green, Cyan, Blue, Violet);
  Suit = (Club, Diamond, Heart, Spade);
```

Each value in the list has an associated ordinality, starting with zero. When you apply the Ord function to a value of an enumerated type, you get this zero-based value. For example, Ord (Diamond) returns 1.

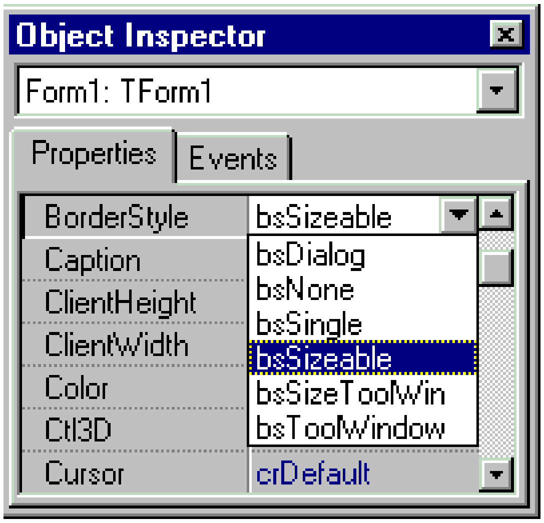
Note: Enumerated types can have different internal representations. By default, Delphi uses an 8-bit representation, unless there are more than 256 different values, in which case it uses the 16-bit representation. There is also a 32-bit representation, which might be useful for compatibility with C or C++ libraries. You can actually change the default behavior, asking for a larger representation, by using the $Z compiler directive.

The Delphi VCL (Visual Component Library) uses enumerated types in many places. For example, the style of the border of a form is defined as follows:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsSizeToolWin, bsToolWindow);
```

When the value of a property is an enumeration, you usually can choose from the list of values displayed in the Object Inspector, as shown in Figure 4.1.

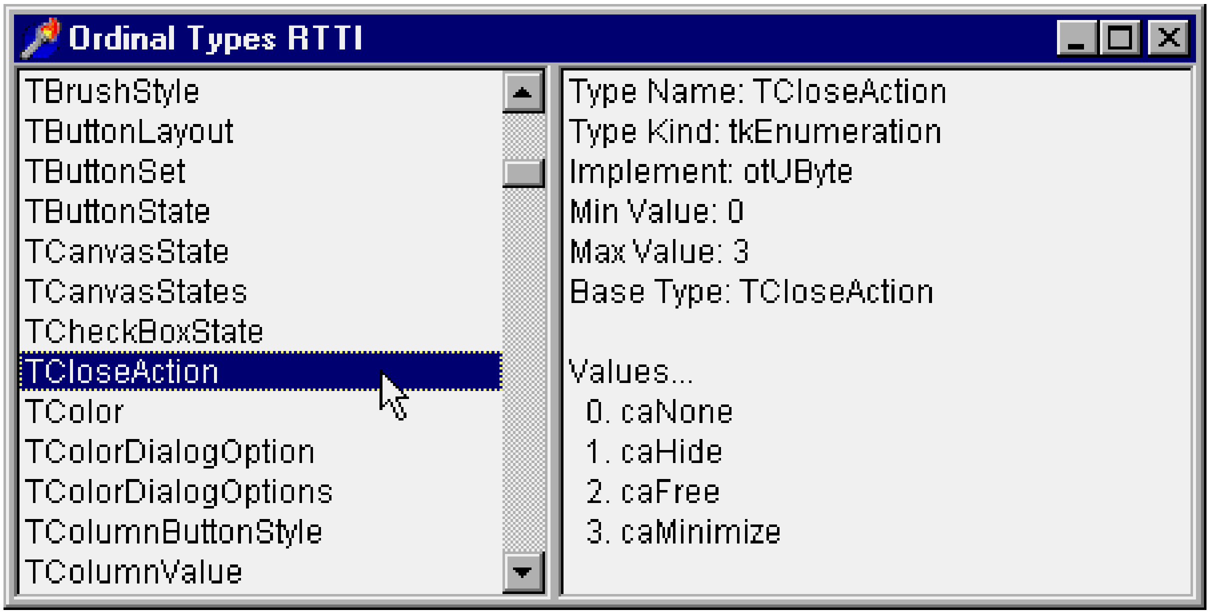
Figure 4.1: An enumerated type property in the Object Inspector



BorderStyle	bsSizeable
Caption	
ClientHeight	
ClientWidth	
Color	
Ctl3D	
Cursor	crDefault

The Delphi Help file generally lists the possible values of an enumeration. As an alternative you can use the OrdType program, available on www.marcocantu.com, to see the list of the values of each Delphi enumeration, set, subrange, and any other ordinal type. You can see an example of the output of this program in Figure 4.2.

Figure 4.2: Detailed information about an enumerated type, as displayed by the OrdType program (available on my web site).



```
Type Name: TCloseAction
Type Kind: tkEnumeration
Implement: otUByte
Min Value: 0
Max Value: 3
Base Type: TCloseAction

Values...
  0. caNone
  1. caHide
  2. caFree
  3. caMinimize
```

Set Types

Set types indicate a group of values, where the list of available values is indicated by the ordinal type the set is based onto. These ordinal types are usually limited, and quite often represented by an enumeration or a subrange. If we take the subrange 1..3, the possible values of the set based on it include only 1, only 2, only 3, both 1 and 2, both 1 and 3, both 2 and 3, all the three values, or none of them.

A variable usually holds one of the possible values of the range of its type. A set-type variable, instead, can contain none, one, two, three, or more values of the range. It can even include all of the values. Here is an example of a set:

```
type
```

```
  Letters = set of Uppercase;
```

Now I can define a variable of this type and assign to it some values of the original type. To indicate some values in a set, you write a comma-separated list, enclosed within square brackets. The following code shows the assignment to a variable of several values, a single value, and an empty value:

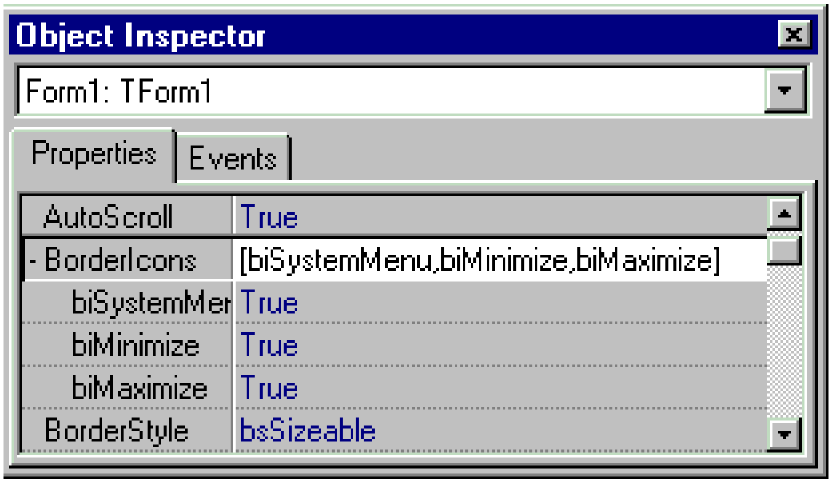
```
var
  Letters1, Letters2, Letters3: Letters;
begin
  Letters1 := ['A', 'B', 'C'];
  Letters2 := ['K'];
  Letters3 := [];
```

In Delphi, a set is generally used to indicate nonexclusive flags. For example, the following two lines of code (which are part of the Delphi library) declare an enumeration of possible icons for the border of a window and the corresponding set type:

```
type
  TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
  TBorderIcons = set of TBorderIcon;
```

In fact, a given window might have none of these icons, one of them, or more than one. When working with the Object Inspector (see Figure 4.3), you can provide the values of a set by expanding the selection (double-click on the property name or click on the plus sign on its left) and toggling on and off the presence of each value.

Figure 4.3: A set-type property in the Object Inspector



AutoScroll	True
- BorderIcons	[biSystemMenu,biMinimize,biMaximize]
biSystemMer	True
biMinimize	True
biMaximize	True
BorderStyle	bsSizeable

Another property based on a set type is the style of a font. Possible values indicate a bold, italic, underline, and strikethrough font. Of course the same font can be both italic and bold, have no attributes, or have them all. For this reason it is declared as a set. You can assign values to this set in the code of a program as follows:

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles
```

You can also operate on a set in many different ways, including adding two variables of the same set type (or, to be more precise, computing the union of the two set variables):

```
Font.Style := OldStyle + [fsUnderline]; // two sets
```

Again, you can use the OrdType examples included in the TOOLS directory of the book source code to see the list of possible values of many sets defined by the Delphi component library.

Array Types

Array types define lists of a fixed number of elements of a specific type. You generally use an index within square brackets to access to one of the elements of the array. The square brackets are used also to specify the possible values of the index when the array is defined. For example, you can define a group of 24 integers with this code:

```
type
  DayTemperatures = array [1..24] of Integer;
```

In the array definition, you need to pass a subrange type within square brackets, or define a new specific subrange type using two constants of an ordinal type. This subrange specifies the valid indexes of the array. Since you specify both the upper and the lower index of the array, the indexes don't need to be zero-based, as is necessary in C, C++, Java, and other programming languages.

Since the array indexes are based on subranges, Delphi can check for their range as we've already seen. An invalid constant subrange results in a compile-time error; and an out-of-range index used at run-time results in a run-time error if the corresponding compiler option is enabled.

Using the array definition above, you can set the value of a DayTemp1 variable of the DayTemperatures type as follows:

```
type
  DayTemperatures = array [1..24] of Integer;

var
  DayTemp1: DayTemperatures;

procedure AssignTemp;
begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;
  DayTemp1 [25] := 67; // compile-time error
```

An array can have more than one dimension, as in the following examples:

```
type
  MonthTemps = array [1..24, 1..31] of Integer;
  YearTemps = array [1..24, 1..31, Jan..Dec] of Integer;
```

These two array types are built on the same core types. So you can declare them using the preceding data types, as in the following code:

```
type
  MonthTemps = array [1..31] of DayTemperatures;
  YearTemps = array [Jan..Dec] of MonthTemps;
```

This declaration inverts the order of the indexes as presented above, but it also allows assignment of whole blocks between variables. For example, the following statement copies January's temperatures to February:

```
var
  ThisYear: YearTemps;
begin
  ...
  ThisYear[Feb] := ThisYear[Jan];
```

You can also define a zero-based array, an array type with the lower bound set to zero. Generally, the use of more logical bounds is an advantage, since you don’t need to use the index 2 to access the third item, and so on. Windows, however, uses invariably zero-based arrays (because it is based on the C language), and the Delphi component library tends to do the same.

If you need to work on an array, you can always test its bounds by using the standard Low and High functions, which return the lower and upper bounds. Using Low and High when operating on an array is highly recommended, especially in loops, since it makes the code independent of the range of the array. Later, you can change the declared range of the array indices, and the code that uses Low and High will still work. If you write a loop hard-coding the range of an array you’ll have to update the code of the loop when the array size changes. Low and High make your code easier to maintain and more reliable.

Note: Incidentally, there is no run-time overhead for using Low and High with arrays. They are resolved at compile-time into constant expressions, not actual function calls. This compile-time resolution of expressions and function calls happens also for many other simple system functions.

Delphi uses arrays mainly in the form of array properties. We have already seen an example of such a property in the TimeNow example, to access the Items property of a ListBox component. I’ll show you some more examples of array properties in the next chapter, when discussing Delphi loops.

Note: Delphi 4 introduced dynamic arrays into Object Pascal , that is arrays that can be resized at runtime allocating the proper amount of memory. Using dynamic arrays is easy, but in this discussion of Pascal I felt they were not an proper topic to cover. You can find a description of Delphi's dynamic arrays in Chapter 8.

Record Types

Record types define fixed collections of items of different types. Each element, or field, has its own type. The definition of a record type lists all these fields, giving each a name you’ll use later to access it.

Here is a small listing with the definition of a record type, the declaration of a variable of that type, and few statements using this variable:

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: Date;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

Classes and objects can be considered an extension of the record type. Delphi libraries tend to use class types instead of record types, but there are many record types defined by the Windows API.

Record types can also have a variant part; that is, multiple fields can be mapped to the same memory area, even if they have a different data type. (This corresponds to a union in the C language.) Alternatively, you can use these variant fields or groups of fields to access the same memory location within a record, but considering those values from different perspectives. The main uses of this type were to store similar but different data and to obtain an effect similar to that of typecasting (something less useful now that typecasting has been introduced also in Pascal). The use of variant record types has been largely replaced by object-oriented and other modern techniques, although Delphi uses them in some peculiar cases.

The use of a variant record type is not type-safe and is not a recommended programming practice, particularly for beginners. Expert programmers can indeed use variant record types, and the core of the Delphi libraries makes use of them. You won't need to tackle them until you are really a Delphi expert, anyway.

Pointers

A pointer type defines a variable that holds the memory address of another variable of a given data type (or an undefined type). So a pointer variable indirectly refers to a value. The definition of a pointer type is not based on a specific keyword, but uses a special character instead. This special symbol is the caret (^):

```
type
  PointerToInt = ^Integer;
```

Once you have defined a pointer variable, you can assign to it the address of another variable of the same type, using the @ operator:

```
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  // change the value in two different ways
  X := 10;
  P^ := 20;
```

When you have a pointer P, with the expression P you refer to the address of the memory location the pointer is referring to, and with the expression P^ you refer to the actual content of that memory location. For this reason in the code fragment above ^P corresponds to X.

Instead of referring to an existing memory location, a pointer can refer to a new memory block dynamically allocated (on the heap memory area) with the New procedure. In this case, when you don't need the pointer any more, you'll also have to to get rid of the memory you've dynamically allocated, by calling the Dispose procedure.

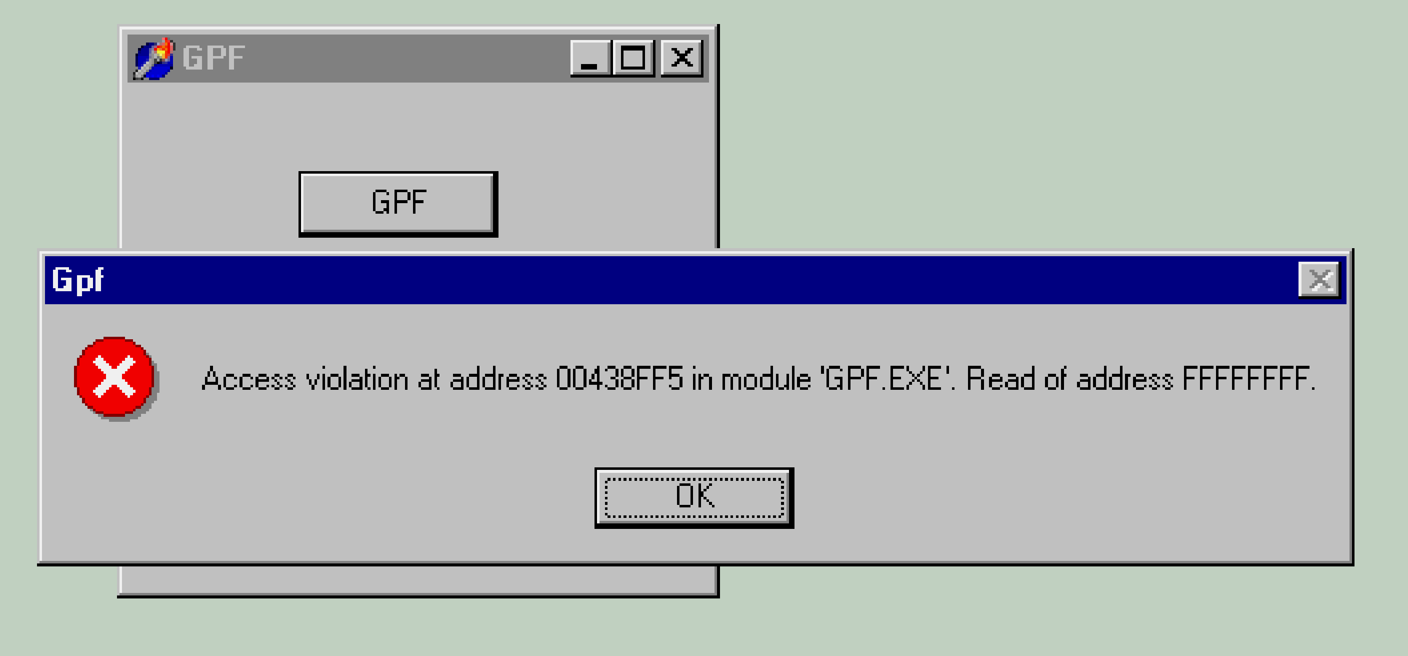
```
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  ShowMessage (IntToStr (P^));
  // termination
  Dispose (P);
end;
```

If a pointer has no value, you can assign the nil value to it. Then you can test whether a pointer is nil to see if it currently refers to a value. This is often used, because dereferencing an invalid pointer causes an access violation (also known as a general protection fault, GPF):

```
procedure TFormGPF.BtnGpfClick(Sender: TObject);
var
  P: ^Integer;
begin
  P := nil;
  ShowMessage (IntToStr (P^));
end;
```

You can see an example of the effect of this code by running the GPF example (or looking at the corresponding Figure 4.4). The example contains also the code fragments shown above.

Figure 4.4: The system error resulting from the access to a nil pointer, from the GPF example.



Access violation at address 00438FF5 in module 'GPF.EXE'. Read of address FFFFFFFF.

In the same program you can find an example of safe data access. In this second case the pointer is assigned to an existing local variable, and can be safely used, but I've added a safe-check anyway:

```
procedure TFormGPF.BtnSafeClick(Sender: TObject);
var
  P: ^Integer;
  X: Integer;
begin
  P := @X;
  X := 100;
  if P <> nil then
    ShowMessage (IntToStr (P^));
end;
```

Delphi also defines a Pointer data type, which indicates untyped pointers (such as void* in the C language). If you use an untyped pointer you should use GetMem instead of New. The GetMem procedure is required each time the size of the memory variable to allocate is not defined.

The fact that pointers are seldom necessary in Delphi is an interesting advantage of this environment. Nonetheless, understanding pointers is important for advanced programming and for a full understanding of the Delphi object model, which uses pointers "behind the scenes."

> Note: Although you don't use pointers often in Delphi, you do frequently use a very similar construct—namely, references. Every object instance is really an implicit pointer or reference to its actual data. However, this is completely transparent to the programmer, who uses object variables just like any other data type.

File Types

Another Pascal-specific type constructor is the file type. File types represent physical disk files, certainly a peculiarity of the Pascal language. You can define a new file data type as follows:

```
type
  IntFile = file of Integer;
```

Then you can open a physical file associated with this structure and write integer values to it or read the current values from the file.

> **Author's Note**: Files-based examples were part of older editions of Mastering Delphi and I plan adding them here as well)

The use of files in Pascal is quite straightforward, but in Delphi there are also some components that are capable of storing or loading their contents to or from a file. There is some serialization support, in the form of streams, and there is also database support.

Conclusion

This chapter discussing user-defined data types complete our coverage of Pascal type system. Now we are ready to look into the statements the language provides to operate on the variables we've defined.

Next Chapter: Statements

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 5 Statements

If the data types are one of the foundations of Pascal programming the other are statements. Statements of the programming language are based on keywords and other elements which allow you to indicate to a program a sequence of operations to perform. Statements are often enclosed in procedures or functions, as we'll see in the next chapter. Now we'll just focus on the basic types of commands you can use to create a program.

Simple and Compound Statements

A Pascal statement is simple when it doesn't contain any other statements. Examples of simple statements are assignment statements and procedure calls. Simple statements are separated by a semicolon:

```
X := Y + Z;  // assignment
Randomize;   // procedure call
```

Usually, statements are part of a compound statement, marked by begin and end brackets. A compound statement can appear in place of a generic Pascal statement. Here is an example:

```
begin
  A := B;
  C := A * 2;
end;
```

The semicolon after the last statement before the end isn't required, as in the following:

```
begin
  A := B;
  C := A * 2
end;
```

Both versions are correct. The first version has a useless (but harmless) semicolon. This semicolon is, in fact, a null statement; that is, a statement with no code. Notice that, at times, null statements can be used inside loops or in other particular cases.

Note: Although these final semicolons serve no purpose, I tend to use them and suggest you do the same. Sometimes after you've written a couple of lines you might want to add one more statement. If the last semicolon is missing you should remember to add it, so it might be better to add it in the first place.

Assignment Statements

Assignments in Pascal use the colon-equal operator, an odd notation for programmers who are used to other languages. The = operator, which is used for assignments in some other languages, in Pascal is used to test for equality.

Note: By using different symbols for an assignment and an equality test, the Pascal compiler (like the C compiler) can translate source code faster, because it doesn't need to examine the context in which the operator is used to determine its meaning. The use of different operators also makes the code easier for people to read.

Conditional Statements

A conditional statement is used to execute either one of the statements it contains or none of them, depending on some test. There are two basic flavors of conditional statements: if statements and case statements.

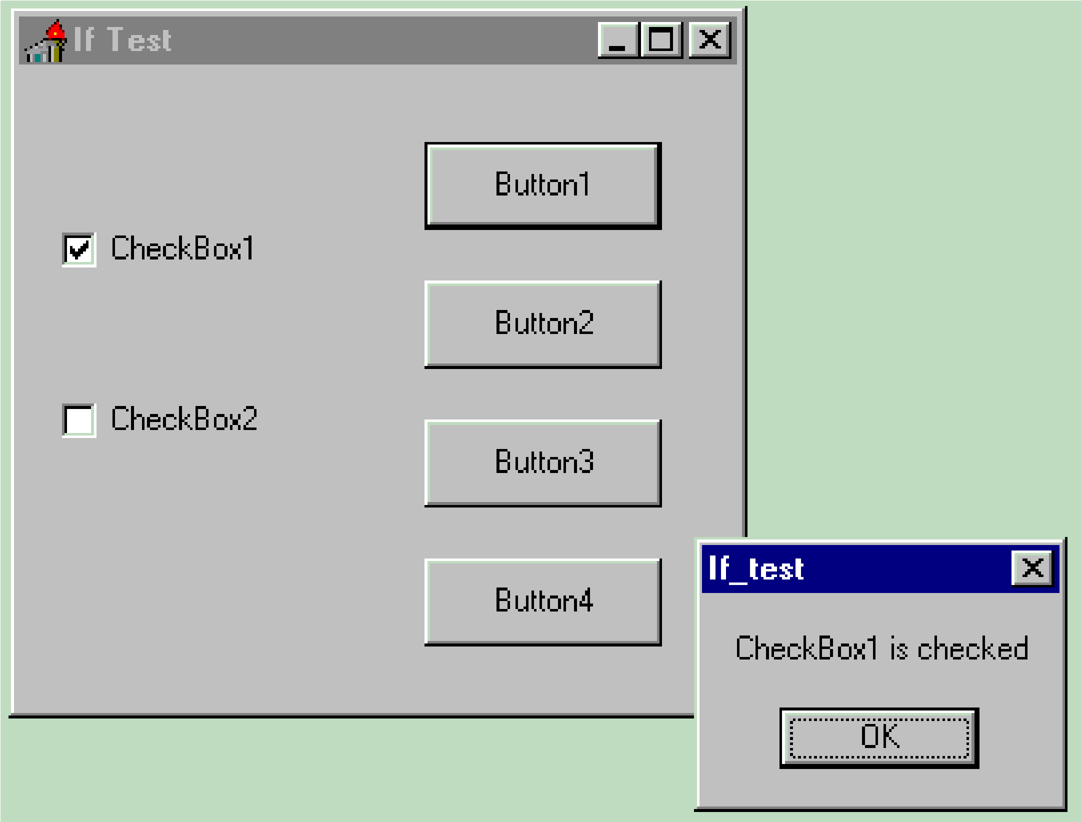
If Statements

The if statement can be used to execute a statement only if a certain condition is met (if-then), or to choose between two different alternatives (if-then-else). The condition is described with a Boolean expression. A simple Delphi example will demonstrate how to write conditional statements. First create a new application, and put two check boxes and four buttons in the form. Do not change the names of buttons or check boxes, but double-click on each button to add a handler for its OnClick event. Here is a simple if statement for the first button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // simple if statement
  if CheckBox1.Checked then
    ShowMessage ('CheckBox1 is checked')
end;
```

When you click on the button, if the first check box has a check mark in it, the program will show a simple message (see Figure 5.1). I've used the ShowMessage function because it is the simplest Delphi function you can use to display a short message to the user.

Figure 5.1: The message displayed by the IfTest example when you press the first button and the first check box is checked.



If you click the button and nothing happens, it means the check box was not checked. In a case like this, it would probably be better to make this more explicit, as with the code for the second button, which uses an if-then-else statement:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  // if-then-else statement
  if CheckBox2.Checked then
    ShowMessage ('CheckBox2 is checked')
  else
    ShowMessage ('CheckBox2 is NOT checked');
end;
```

Notice that you cannot have a semicolon after the first statement and before the else keyword, or the compiler will issue a syntax error. The if-then-else statement, in fact, is a single statement, so you cannot place a semicolon in the middle of it.

An if statement can be quite complex. The condition can be turned into a series of conditions (using the and, or and not Boolean operators), or the if statement can nest a second if statement. The last two buttons of the IfTest example demonstrate these cases:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  // statement with a double condition
  if CheckBox1.Checked and CheckBox2.Checked then
    ShowMessage ('Both check boxes are checked')
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  // compound if statement
  if CheckBox1.Checked then
    if CheckBox2.Checked then
      ShowMessage ('CheckBox1 and 2 are checked')
    else
      ShowMessage ('Only CheckBox1 is checked')
  else
    ShowMessage (
      'Checkbox1 is not checked, who cares for Checkbox2?')
end;
```

Look at the code carefully and run the program to see if you understand everything. When you have doubts about a programming construct, writing a very simple program such as this can help you learn a lot. You can add more check boxes and increase the complexity of this small example, making any test you like.

Case Statements

If your if statements become very complex, at times you can replace them with case statements. A case statement consists in an expression used to select a value, a list of possible values, or a range of values. These values are constants, and they must be unique and of an ordinal type. Eventually, there can be an else statement that is executed if none of the labels correspond to the value of the selector. Here are two simple examples:

```
case Number of
  1: Text := 'One';
  2: Text := 'Two';
  3: Text := 'Three';
end;

case MyChar of
  '+' : Text := 'Plus sign';
  '-' : Text := 'Minus sign';
  '*', '/': Text := 'Multiplication or division';
  '0'..'9': Text := 'Number';
  'a'..'z': Text := 'Lowercase character';
  'A'..'Z': Text := 'Uppercase character';
else
  Text := 'Unknown character';
end;
```

Loops in Pascal

The Pascal language has the typical repetitive statements of most programming languages, including for, while, and repeat statements. Most of what these loops do will be familiar if you've used other programming languages, so I'll cover them only briefly.

The For Loop

The for loop in Pascal is strictly based on a counter, which can be either increased or decreased each time the loop is executed. Here is a simple example of a for loop used to add the first ten numbers.

```
var
  K, I: Integer;
begin
  K := 0;
  for I := 1 to 10 do
    K := K + I;
```

This same for statement could have been written using a reverse counter:

```
var
  K, I: Integer;
begin
  K := 0;
  for I := 10 downto 1 do
    K := K + I;
```

The for loop in Pascal is less flexible than in other languages (it is not possible to specify an increment different than one), but it is simple and easy to understand. If you want to test for a more complex condition, or to provide a customized counter, you need to use a while or repeat statement, instead of a for loop.

Note: The counter of a for loop doesn't need to be a number. It can be a value of any ordinal type, such as a character or an enumerated type.

While and Repeat Statements

The difference between the while-do loop and the repeat-until loop is that the code of the repeat statement is always executed at least once. You can easily understand why by looking at a simple example:

```
while (I <= 100) and (J <= 100) do
begin
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
end;

repeat
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

If the initial value of I or J is greater than 100, the statements inside the repeat-until loop are executed once anyway.

The other key difference between these two loops is that the repeat-until loop has a reversed condition. The loop is executed as long as the condition is not met. When the condition is met, the loop terminates. This is the opposite from a while-do loop, which is executed while the condition is true. For this reason I had to reverse the condition in the code above to obtain a similar statement.

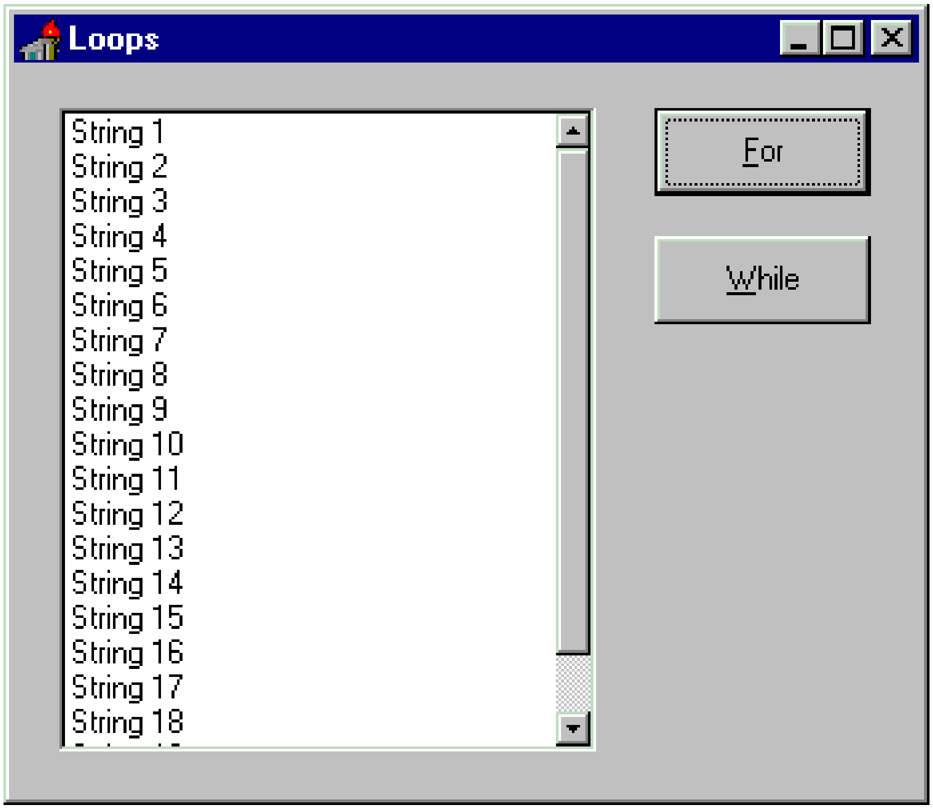
An Example of Loops

To explore the details of loops, let's look at a small Delphi example. The Loops program highlights the difference between a loop with a fixed counter and a loop with an almost random counter. Start with a new blank project, place a list box and two buttons on the main form, and give the buttons a proper name (BtnFor and BtnWhile) by setting their Name property in the Object Inspector. You can also remove the word Btn from the Caption property (and eventually even add the & character to it to activate the following letter as a shortcut key). Here is a summary of the textual description of this form:

```
object Form1: TForm1
  Caption = 'Loops'
  object ListBox1: TListBox ...
  object BtnFor: TButton
```

```
    Caption = '&For'
    OnClick = BtnForClick
  end
  object BtnWhile: TButton
    Caption = '&While'
    OnClick = BtnWhileClick
  end
end
```

Figure 5.2: Each time you press the For button of the Loops example, the list box is filled with consecutive numbers.



```
String 1
String 2
String 3
String 4
String 5
String 6
String 7
String 8
String 9
String 10
String 11
String 12
String 13
String 14
String 15
String 16
String 17
String 18
```

Now we can add some code to the OnClick events of the two buttons. The first button has a simple for loop to display a list of numbers, as you can see in Figure 5.2. Before executing this loop, which adds a number of strings to the Items property of the list box, you need to clear the contents of the list box itself:

```
procedure TForm1.BtnForClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 1 to 20 do
    Listbox1.Items.Add ('String ' + IntToStr (I));
end;
```

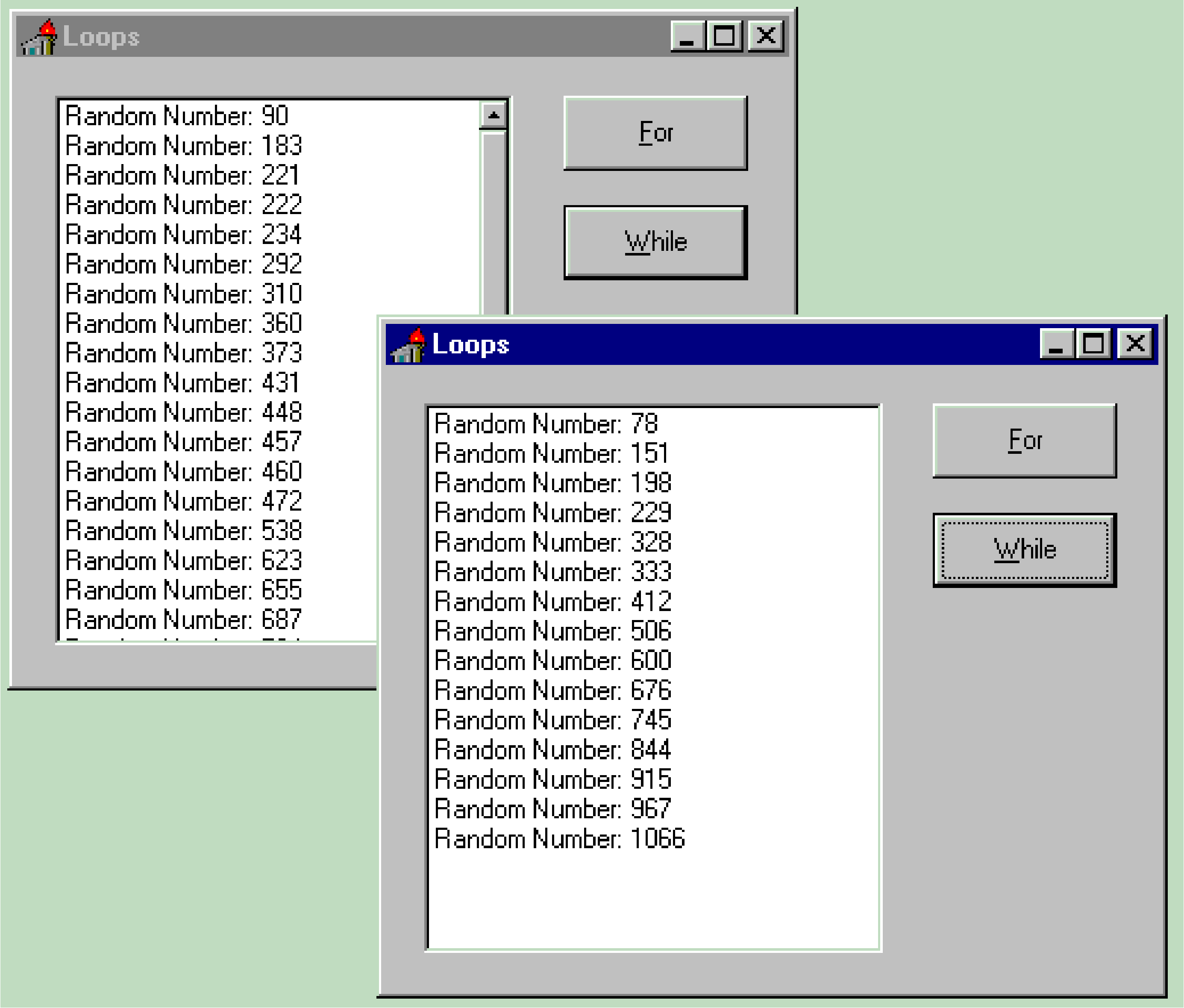
The code associated with the second button is slightly more complex. In this case, there is a while loop based on a counter, which is increased randomly. To accomplish this, I've called the Randomize procedure, which resets the random number generator, and the Random function with a range value of 100. The result of this function is a number between 0 and 99, chosen randomly. The series of random numbers control how many times the while loop is executed.

```
procedure TForm1.BtnWhileClick(Sender: TObject);
var
```

```
  I: Integer;
begin
  ListBox1.Items.Clear;
  Randomize;
  I := 0;
  while I < 1000 do
  begin
    I := I + Random (100);
    Listbox1.Items.Add ('Random Number: ' + IntToStr (I));
  end;
end;
```

Each time you click the While button, the numbers are different, because they depend on the random-number generator. Figure 5.3 shows the results from two separate button-clicks. Notice that not only are the generated numbers different each time, but so is the number of items. That is, this while loop is executed a random numbers of times. If you press the While button several times in a row, you'll see that the list box has a different number of lines.

Figure 5.3: The contents of the list box of the Loops example change each time you press the While button. Because the loop counter is incremented by a random value, every time you press the button the loop may execute a different number of times.



```
Random Number: 90
Random Number: 183
Random Number: 221
Random Number: 222
Random Number: 234
Random Number: 292
Random Number: 310
Random Number: 360
Random Number: 373
Random Number: 431
Random Number: 448
Random Number: 457
Random Number: 460
Random Number: 472
Random Number: 538
Random Number: 623
Random Number: 655
Random Number: 687
```

```
Random Number: 78
Random Number: 151
Random Number: 198
Random Number: 229
Random Number: 328
Random Number: 333
Random Number: 412
Random Number: 506
Random Number: 600
Random Number: 676
Random Number: 745
Random Number: 844
Random Number: 915
Random Number: 967
Random Number: 1066
```

Note: You can alter the standard flow of a loop's execution using the Break and Continue system procedures. The first interrupts the loop; the second is used to jump directly to the loop test or counter increment, continuing with the next iteration of the loop (unless the condition is zero or the counter has reached its highest value). Two more system procedures, Exit and Halt, let you immediately return from the current function or procedure or terminate the program.

The With Statement

The last kind of Pascal statement I'll focus on is the with statement, which used to be peculiar to this programming language (although it has been recentrly introduced also in JavaScript and Visual Basic) and can be very useful in Delphi programming.

The with statement is nothing but shorthand. When you need to refer to a record type variable (or an object), instead of repeating its name every time, you can use a with statement. For example, while presenting the record type, I wrote this code:

```
type
  Date = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: Date;

begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

Using a with statement, I can improve the final part of this code, as follows:

```
begin
  with BirthDay do
  begin
    Year := 1995;
    Month := 2;
    Day := 14;
  end;
```

This approach can be used in Delphi programs to refer to components and other class types. For example, we can rewrite the final part of the last example, Loops, using a with statement to access the items of the list box:

```
procedure TForm1.WhileButtonClick(Sender: TObject);
var
  I: Integer;
begin
  with ListBox1.Items do
  begin
    Clear; // shortcut
    Randomize;
    I := 0;
    while I < 1000 do
    begin
      I := I + Random (100);
      // shortcut:
```

```
      Add ('Random Number: ' + IntToStr (I));
    end;
  end;
end;
```

When you work with components or classes in general, the with statement allows you to skip writing some code, particularly for nested fields. For example, suppose that you need to change the Width and the Color of the drawing pen for a form. You can write the following code:

```
Form1.Canvas.Pen.Width := 2;
Form1.Canvas.Pen.Color := clRed;
```

But it is certainly easier to write this code:

```
with Form1.Canvas.Pen do
begin
  Width := 2;
  Color := clRed;
end;
```

When you are writing complex code, the with statement can be effective and spares you the declaration of some temporary variables, but it has a drawback. It can make the code less readable, particularly when you are working with different objects that have similar or corresponding properties.

A further drawback is that using the with statement can allow subtle logical errors in the code that the compiler will not detect. For example:

```
with Button1 do
begin
  Width := 200;
  Caption := 'New Caption';
  Color := clRed;
end;
```

This code changes the Caption and the Width of the button, but it affects the Color property of the form, not that of the button! The reason is that the TButton components don't have the Color property, and since the code is executed for a form object (we are writing a method of the form) this object is accessed by default. If we had instead written:

```
Button1.Width := 200;
Button1.Caption := 'New Caption';
Button1.Color := clRed; // error!
```

the compiler would have issued an error. In general, we can say that since the with statement introduces new identifiers in the current scope, we might hide existing identifiers, or wrongfully access another identifier in the same scope (as in the first version of this code fragment). Even considering this kind of drawback, I suggest you get used to with statements, because they can be really very handy, and at times even make the code more readable.

You should, however, avoid using multiple with statements, such as:

```
with ListBox1, Button1 do...
```

The code following this would probably be highly unreadable, because for each property defined in this block you would need to think about which component it refers to, depending on the respective properties and the order of the components in the with statement.

Note: Speaking of readability, Pascal has no endif or endcase statement. If an if statement has a begin-end block, then the end of the block marks the end of the statement. The case statement, instead, is always terminated by an end. All these end statements, often found one after the other, can make the code difficult to follow. Only by tracing the indentations can you see which statement a particular end refers to. A common way to solve this problem and make the code more readable is to add a comment after the end statement indicating its role, as in:

```
if ... then
 ...
end; // if
```

Conclusion

This chapter has described how to code conditional statements and loops. Instead of writing long lists of such statements, programs are usually split in routines, procedures or functions. This is the topic of the next chapter, which introduces also some advanced elements of Pascal.

Next Chapter: Procedures

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 6 Procedures and Functions

Another important idea emphasized by Pascal is the concept of the routine, basically a series of statements with a unique name, which can be activated many times by using their name. This way you avoid repeating the same statements over and over, and having a single version of the code you can easily modify it all over the program. From this point of view, you can think of routines as the basic code encapsulation mechanism. I'll get back to this topic with an example after I introduce the Pascal routines syntax.

Pascal Procedures and Functions

In Pascal, a routine can assume two forms: a procedure and a function. In theory, a procedure is an operation you ask the computer to perform, a function is a computation returning a value. This difference is emphasized by the fact that a function has a result, a return value, while a procedure doesn't. Both types of routines can have multiple parameters, of given data types.

In practice, however, the difference between functions and procedures is very limited: you can call a function to perform some work and then skip the result (which might be an optional error code or something like that) or you can call a procedure which passes a result within its parameters (more on reference parameters later in this chapter).

Here are the definitions of a procedure and two versions of the same function, using a slightly different syntax:

```
procedure Hello;
begin
  ShowMessage ('Hello world!');
end;

function Double (Value: Integer) : Integer;
begin
  Double := Value * 2;
end;

// or, as an alternative
function Double2 (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

The use of Result instead of the function name to assign the return value of a function is becoming quite popular, and tends to make the code more readable, in my opinion.

Once these routines have been defined, you can call them one or more times. You call the procedure to make it perform its task, and call a function to compute the value:

```
procedure TForm1.Button1Click (Sender: TObject);
begin
```

```
  Hello;
end;

procedure TForm1.Button2Click (Sender: TObject);
var
  X, Y: Integer;
begin
  X := Double (StrToInt (Edit1.Text));
  Y := Double (X);
  ShowMessage (IntToStr (Y));
end;
```

Note: For the moment don't care about the syntax of the two procedures above, which are actually methods. Simply place two buttons on a Delphi form, click on them at design time, and the Delphi IDE will generate the proper support code: Now you simply have to fill in the lines between begin and end. To compile the code above you need to add also an Edit control to the form.

Now we can get back to the encapsulation code concept I've introduced before. When you call the Double function, you don't need to know the algorithm used to implement it. If you later find out a better way to double numbers, you can easily change the code of the function, but the calling code will remain unchanged (although executing it will be faster!). The same principle can be applied to the Hello procedure: We can modify the program output by changing the code of this procedure, and the Button2Click method will automatically change its effect. Here is how we can change the code:

```
procedure Hello;
begin
  MessageDlg ('Hello world!', mtInformation, [mbOK]);
end;
```

Tip: When you call an existing Delphi function or procedure, or any VCL method, you should remember the number and type of the parameters. Delphi editor helps you by suggesting the parameters list of a function or procedure with a fly-by hint as soon as you type its name and the open parenthesis. This feature is called Code Parameters and is part of the Code Insight technology.

Reference Parameters

Pascal routines allow parameter passing by value and by reference. Passing parameters by value is the default: the value is copied on the stack and the routine uses and manipulates the copy, not the original value.

Passing a parameter by reference means that its value is not copied onto the stack in the formal parameter of the routine (avoiding a copy often means that the program executes faster). Instead, the program refers to the original value, also in the code of the routine. This allows the procedure or function to change the value of the parameter. Parameter passing by reference is expressed by the var keyword.

This technique is available in most programming languages. It isn't present in C, but has been introduced in C++, where you use the & (pass by reference) symbol. In Visual Basic every parameter not specified as ByVal is passed by reference.

Here is an example of passing a parameter by reference using the var keyword:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

In this case, the parameter is used both to pass a value to the procedure and to return a new value to the calling code. When you write:

```
var
  X: Integer;
begin
  X := 10;
  DoubleTheValue (X);
```

the value of the X variable becomes 20, because the function uses a reference to the original memory location of X, affecting its initial value.

Passing parameters by reference makes sense for ordinal types, for old-fashioned strings, and for large records. Delphi objects, in fact, are invariably passed by value, because they are references themselves. For this reason passing an object by reference makes little sense (apart from very special cases), because it corresponds to passing a "reference to a reference."

Delphi long strings have a slightly different behavior: they behave as references, but if you change one of the string variables referring to the same string in memory, this is copied before updating it. A long string passed as a value parameter behaves as a reference only in terms of memory usage and speed of the operation. But if you modify the value of the string, the original value is not affected. On the contrary, if you pass the long string by reference, you can alter the original value.

Delphi 3 introduced a new kind of parameter, out. An out parameter has no initial value and is used only to return a value. These parameters should be used only for COM procedures and functions; in general, it is better to stick with the more efficient var parameters. Except for not having an initial value, out parameters behave like var parameters.

Constant Parameters

As an alternative to reference parameters, you can use a const parameter. Since you cannot assign a new value to a constant parameter inside the routine, the compiler can optimize parameter passing. The compiler can choose an approach similar to reference parameters (or a const reference in C++ terms), but the behavior will remain similar to value parameters, because the original value won't be affected by the routine.

In fact, if you try to compile the following (silly) code, Delphi will issue an error:

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;      // compiler error
  Result := Value;
end;
```

Open Array Parameters

Unlike C, a Pascal function or procedure always has a fixed number of parameters. However, there is a way to pass a varying number of parameters to a routine using an open array.

The basic definition of an open array parameter is that of a typed open array. This means you indicate the type of the parameter but do not know how many elements of that type the array is going to have. Here is an example of such a definition:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

Using High(A) we can get the size of the array. Notice also the use of the return value of the function, Result, to store temporary values. You can call this function by passing to it an array of Integer expressions:

```
X := Sum ([10, Y, 27*I]);
```

Given an array of Integers, of any size, you can pass it directly to a routine requiring an open array parameter or, instead, you can call the Slice function to pass only a portion of the array (as indicated by its second parameter). Here is an example, where the complete array is passed as parameter:

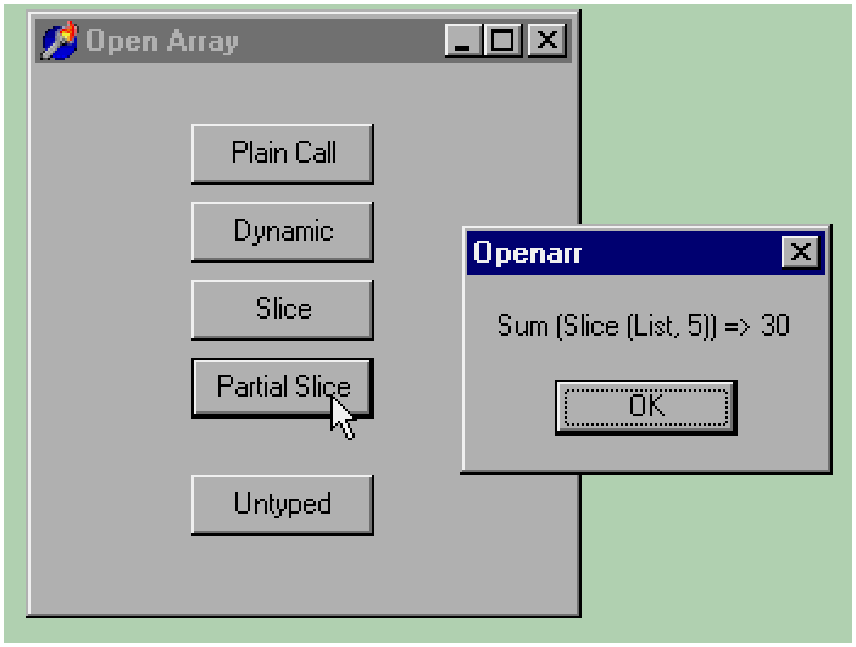
```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);
```

If you want to pass only a portion of the array to the Slice function, simply call it this way:

```
X := Sum (Slice (List, 5));
```

You can find all the code fragments presented in this section in the OpenArr example (see Figure 6.1, later on, for the form).

Figure 6.1: The OpenArr example when the Partial Slice button is pressed



Sum (Slice (List, 5)) => 30

> Typed open arrays in Delphi 4 are fully compatible with dynamic arrays (introduced in Delphi 4 and covered in Chapter 8). Dynamic arrays use the same syntax as open arrays, with the difference that you can use a notation such as array of Integer to declare a variable, not just to pass a parameter.

Type-Variant Open Array Parameters

Besides these typed open arrays, Delphi allows you to define type-variant or untyped open arrays. This special kind of array has an undefined number of values, which can be handy for passing parameters.

Technically, the construct array of const allows you to pass an array with an undefined number of elements of different types to a routine at once. For example, here is the definition of the Format function (we'll see how to use this function in **Chapter 7**, covering strings):

```
function Format (const Format: string;
  const Args: array of const): string;
```

The second parameter is an open array, which gets an undefined number of values. In fact, you can call this function in the following ways:

```
N := 20;
S := 'Total:';
Label1.Caption := Format ('Total: %d', [N]);
Label2.Caption := Format ('Int: %d, Float: %f', [N, 12.4]);
Label3.Caption := Format ('%s %d', [S, N * 2]);
```

Notice that you can pass a parameter as either a constant value, the value of a variable, or an expression. Declaring a function of this kind is simple, but how do you code it? How do you know the types of the parameters? The values of a type-variant open array parameter are compatible with the TVarRec type elements.

Note: Do not confuse the TVarRec record with the TVarData record used by the Variant type itself. These two structures have a different aim and are not compatible. Even the list of possible types is different, because TVarRec can hold Delphi data types, while TVarData can hold OLE data types.

The TVarRec record has the following structure:

```
type
  TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
  end;
```

Each possible record has the VType field, although this is not easy to see at first because it is declared only once, along with the actual Integer-size data (generally a reference or a pointer).

Using this information we can actually write a function capable of operating on different data types. In the SumAll function example, I want to be able to sum values of different types, transforming strings to integers, characters to the corresponding order value, and adding 1 for True Boolean values. The code is based on a case statement, and is quite simple, although we have to dereference pointers quite often:

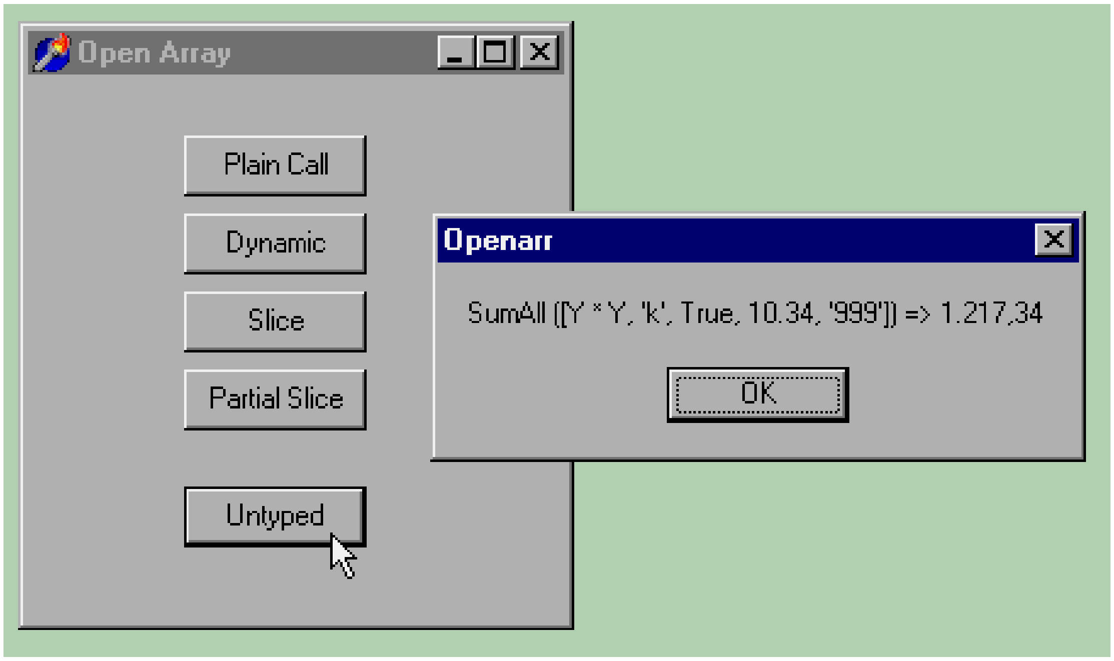
```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger: Result :=
        Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtChar:
        Result := Result + Ord (Args [I].VChar);
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtString, vtAnsiString:
        Result := Result + StrToIntDef ((Args [I].VString^), 0);
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
end;
```

I've added this code to the OpenArr example, which calls the SumAll function when a given button is pressed:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34, '99999']);
  ShowMessage (Format (
    'SumAll ([Y*Y, ''k'', True, 10.34, ''99999'']) => %n', [X]));
end;
```

You can see the output of this call, and the form of the OpenArr example, in Figure 6.2.

Figure 6.2: The form of the OpenArr example, with the message box displayed when the Untyped button is pressed.



SumAll ([Y * Y, 'k', True, 10.34, '999']) => 1.217,34

Delphi Calling Conventions

The 32-bit version of Delphi has introduced a new approach to passing parameters, known as fastcall: Whenever possible, up to three parameters can be passed in CPU registers, making the function call much faster. The fast calling convention (used by default in Delphi 3) is indicated by the register keyword.

The problem is that this is the default convention, and functions using it are not compatible with Windows: the functions of the Win32 API must be declared using the stdcall calling convention, a mixture of the original Pascal calling convention of the Win16 API and the cdecl calling convention of the C language.

There is generally no reason not to use the new fast calling convention, unless you are making external Windows calls or defining Windows callback functions. We'll see an example using the stdcall convention before the end of this chapter. You can find a summary of Delphi calling conventions in the Calling conventions topic under Delphi help.

What Is a Method?

If you have already worked with Delphi or read the manuals, you have probably heard about the term "method". A method is a special kind of function or procedure that is related to a data type, a class. In Delphi, every time we handle an event, we need to define a method, generally a procedure. In general, however, the term method is used to indicate both functions and procedures related to a class.

We have already seen a number of methods in the examples in this and the previous chapters. Here is an empty method automatically added by Delphi to the source code of a form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {here goes your code}
end;
```

Forward Declarations

When you need to use an identifier (of any kind), the compiler must have already seen some sort of declaration to know what the identifier refers to. For this reason, you usually provide a full declaration before using any routine. However, there are cases in which this is not possible. If procedure A calls procedure B, and procedure B calls procedure A, when you start writing the code, you will need to call a routine for which the compiler still hasn't seen a declaration.

If you want to declare the existence of a procedure or function with a certain name and given parameters, without providing its actual code, you can write the procedure or function followed by the forward keyword:

```
procedure Hello; forward;
```

Later on, the code should provide a full definition of the procedure, but this can be called even before it is fully defined. Here is a silly example, just to give you the idea:

```
procedure DoubleHello; forward;

procedure Hello;
begin
  if MessageDlg ('Do you want a double message?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    DoubleHello
  else
    ShowMessage ('Hello');
end;

procedure DoubleHello;
begin
  Hello;
  Hello;
end;
```

This approach allows you to write mutual recursion: DoubleHello calls Hello, but Hello might call DoubleHello, too. Of course there must be a condition to terminate the recursion, to avoid a stack overflow. You can find this code, with some slight changes, in the DoubleH example.

Although a forward procedure declaration is not very common in Delphi, there is a similar case that is much more frequent. When you declare a procedure or function in the interface portion of a unit (more on units in the next chapter), it is considered a forward declaration, even if the forward keyword is not present. Actually you cannot write the body of a routine in the interface portion of a unit. At the same time, you must provide in the same unit the actual implementation of each routine you have declared.

The same holds for the declaration of a method inside a class type that was automatically generated by Delphi (as you added an event to a form or its components). The event handlers declared inside a TForm class are forward declarations: the code will be provided in the implementation portion of the unit. Here is an excerpt of the source code of an earlier example, with the declaration of the Button1Click method:

```
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;
```

Procedural Types

Another unique feature of Object Pascal is the presence of procedural types. These are really an advanced language topic, which only a few Delphi programmers will use regularly. However, since we will discuss related topics in later chapters (specifically, method pointers, a technique heavily used by Delphi), it's worth a quick look at them here. If you are a novice programmer, you can skip this section for now, and come back to it when you feel ready.

In Pascal, there is the concept of procedural type (which is similar to the C language concept of function pointer). The declaration of a procedural type indicates the list of parameters and, in the case of a function, the return type. For example, you can declare a new procedural type, with an Integer parameter passed by reference, with this code:

```
type
  IntProc = procedure (var Num: Integer);
```

This procedural type is compatible with any routine having exactly the same parameters (or the same function signature, to use C jargon). Here is an example of a compatible routine:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

Note: In the 16-bit version of Delphi, routines must be declared using the far directive in order to be used as actual values of a procedural type.

Procedural types can be used for two different purposes: you can declare variables of a procedural type or pass a procedural type (that is, a function pointer) as parameter to another routine. Given the preceding type and procedure declarations, you can write this code:

```
var
  IP: IntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

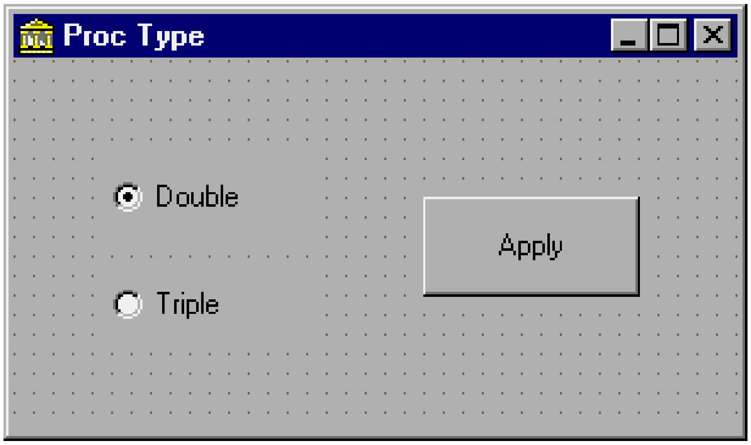
This code has the same effect as the following shorter version:

```
var
  X: Integer;
begin
  X := 5;
  DoubleTheValue (X);
end;
```

The first version is clearly more complex, so why should we use it? In some cases, being able to decide which function to call and actually calling it later on can be useful. It is possible to build a complex example showing this approach. However, I prefer to let you explore a fairly simple one, named ProcType. This example is more complex than those we have seen so far, to make the situation a little more realistic.

Simply create a blank project and place two radio buttons and a push button, as shown in Figure 6.3. This example is based on two procedures. One procedure is used to double the value of the parameter. This procedure is similar to the version I've already shown in this section. A second procedure is used to triple the value of the parameter, and therefore is named TripleTheValue:

Figure 6.3: The form of the ProcType example.



```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
  ShowMessage ('Value tripled: ' + IntToStr (Value));
end;
```

Both procedures display what is going on, to let us know that they have been called. This is a simple debugging feature you can use to test whether or when a certain portion of code is executed, instead of adding a breakpoint in it.

Each time a user presses the Apply button, one of the two procedures is executed, depending on the status of the radio buttons. In fact, when you have two radio buttons in a form, only one of them can be selected at a time. This code could have been implemented by testing the value of the radio buttons inside the code for the OnClick event of the Apply button. To demonstrate the use of procedural types, I've instead used a longer but interesting approach. Each time a user clicks on one of the two radio buttons, one of the procedures is stored in a variable:

```
procedure TForm1.DoubleRadioButtonClick(Sender: TObject);
begin
  IP := DoubleTheValue;
end;
```

When the user clicks on the push button, the procedure we have stored is executed:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  IP (X);
end;
```

To allow three different functions to access the IP and X variables, we need to make them visible to the whole form; they cannot be declared locally (inside one of the methods). A solution to this problem is to place these variables inside the form declaration:

```
type
  TForm1 = class(TForm)
    ...
  private
    { Private declarations }
    IP: IntProc;
    X: Integer;
  end;
```

We will see exactly what this means in the next chapter, but for the moment, you need to modify the code generated by Delphi for the class type as indicated above, and add the definition of the procedural type I've shown before. To initialize these two variables with suitable values, we can handle the OnCreate event of the form (select this event in the Object Inspector after you have activated the form, or simply double-click on the form). I suggest you refer to the listing to study the details of the source code of this example.

You can see a practical example of the use of procedural types in Chapter 9, in the section A Windows Callback Function.

Function Overloading

The idea of overloading is simple: The compiler allows you to define two functions or procedures using the same name, provided that the parameters are different. By checking the parameters, in fact, the compiler can determine which of the versions of the routine you want to call.

Consider this series of functions extracted from the Math unit of the VCL:

```
function Min (A,B: Integer): Integer; overload;
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;
```

When you call Min (10, 20), the compiler easily determines that you're calling the first function of the group, so the return value will be an Integer.

The basic rules are two:

- Each version of the routine must be followed by the overload keyword.
- The differences must be in the number or type of the parameters, or both. The return type, instead, cannot be used to distinguish among two routines.

Here are three overloaded versions of a ShowMsg procedure I've added to the OverDef example (an application demonstrating overloading and default parameters):

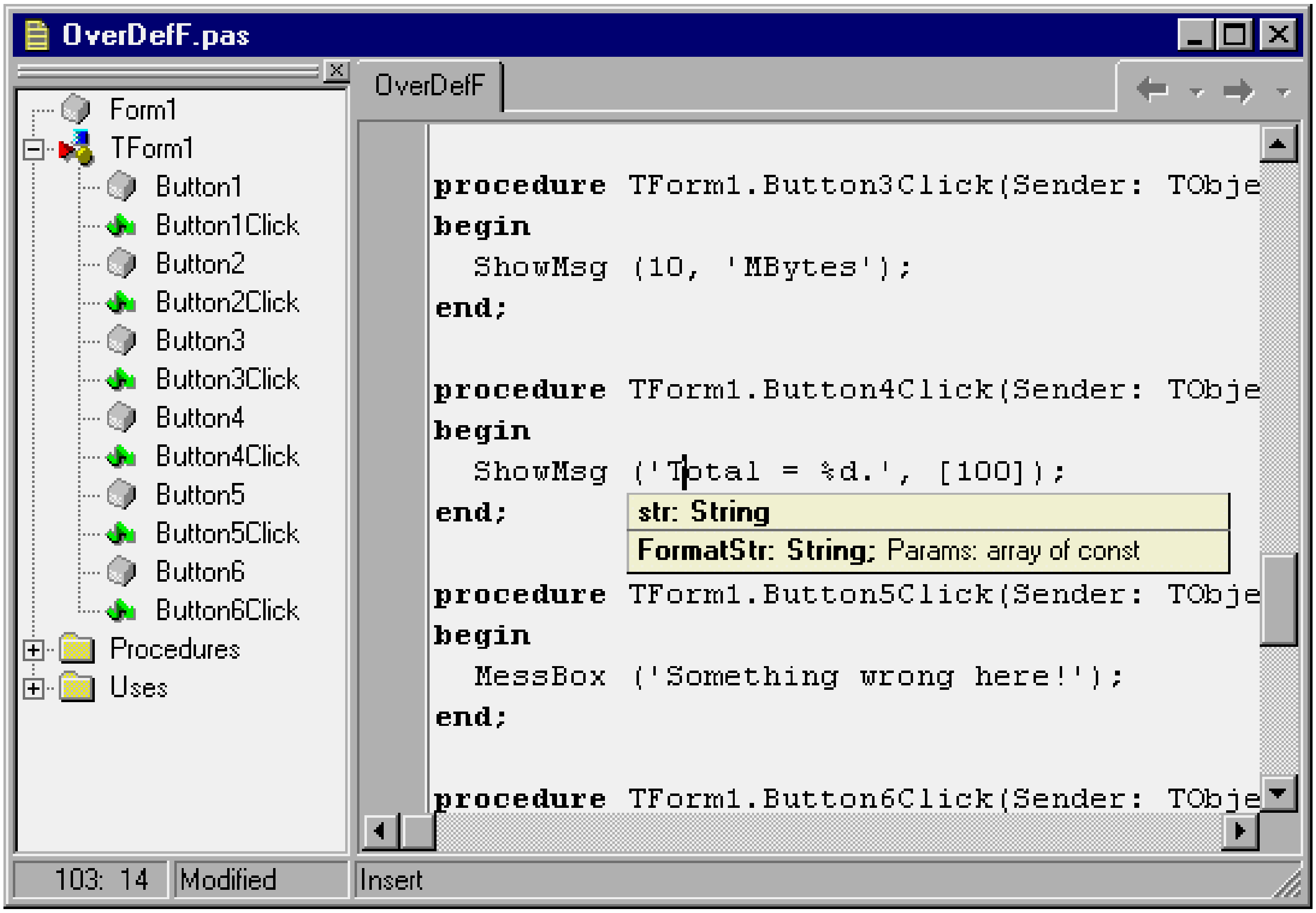
```
procedure ShowMsg (str: string); overload;
begin
  MessageDlg (str, mtInformation, [mbOK], 0);
end;

procedure ShowMsg (FormatStr: string;
  Params: array of const); overload;
begin
  MessageDlg (Format (FormatStr, Params),
    mtInformation, [mbOK], 0);
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
  ShowMsg (IntToStr (I) + ' ' + Str);
end;
```

The three functions show a message box with a string, after optionally formatting the string in different ways. Here are the three calls of the program:

```
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

What surprised me in a positive way is that Delphi's Code Parameters technology works very nicely with overloaded procedures and functions. As you type the open parenthesis after the routine name, all the available alternatives are listed. As you enter the parameters, Delphi uses their type to determine which of the alternatives are still available. In Figure 6.4 you can see that after starting to type a constant string Delphi shows only the compatible versions (omitting the version of the ShowMsg procedure that has an integer as first parameter).

Figure 6.4: The multiple alternatives offered by Code Parameters for overloaded routines are filtered according to the parameters already available.



```
procedure TForm1.Button3Click(Sender: TObje
begin
  ShowMsg (10, 'MBytes');
end;

procedure TForm1.Button4Click(Sender: TObje
begin
  ShowMsg ('Total = %d.', [100]);
end;       str: String
           FormatStr: String; Params: array of const

procedure TForm1.Button5Click(Sender: TObje
begin
  MessBox ('Something wrong here!');
end;

procedure TForm1.Button6Click(Sender: TObje
```

The fact that each version of an overloaded routine must be properly marked implies that you cannot overload an existing routine of the same unit that is not marked with the overload keyword. (The error message you get when you try is: "Previous declaration of '<name>' was not marked with the 'overload' directive.") However, you can overload a routine that was originally declared in a different unit. This is for compatibility with previous versions of Delphi, which allowed different units to reuse the same routine name. Notice, anyway, that this special case is not an extra feature of overloading, but an indication of the problems you can face.

For example, you can add to a unit the following code:

```
procedure MessageDlg (str: string); overload;
begin
  Dialogs.MessageDlg (str, mtInformation, [mbOK], 0);
end;
```

This code doesn't really overload the original MessageDlg routine. In fact if you write:

```
MessageDlg ('Hello');
```

you'll get a nice error message indicating that some of the parameters are missing. The only way to call the local version instead of the one of the VCL is to refer explicitly to the local unit, something that defeats the idea of overloading:

```
OverDefF.MessageDlg ('Hello');
```

Default Parameters

A related new feature of Delphi 4 is that you can give a default value for the parameter of a function, and you can call the function with or without the parameter. Let me show an example. We can define the following encapsulation of the MessageBox method of the Application global object, which uses PChar instead of strings, providing two default parameters:

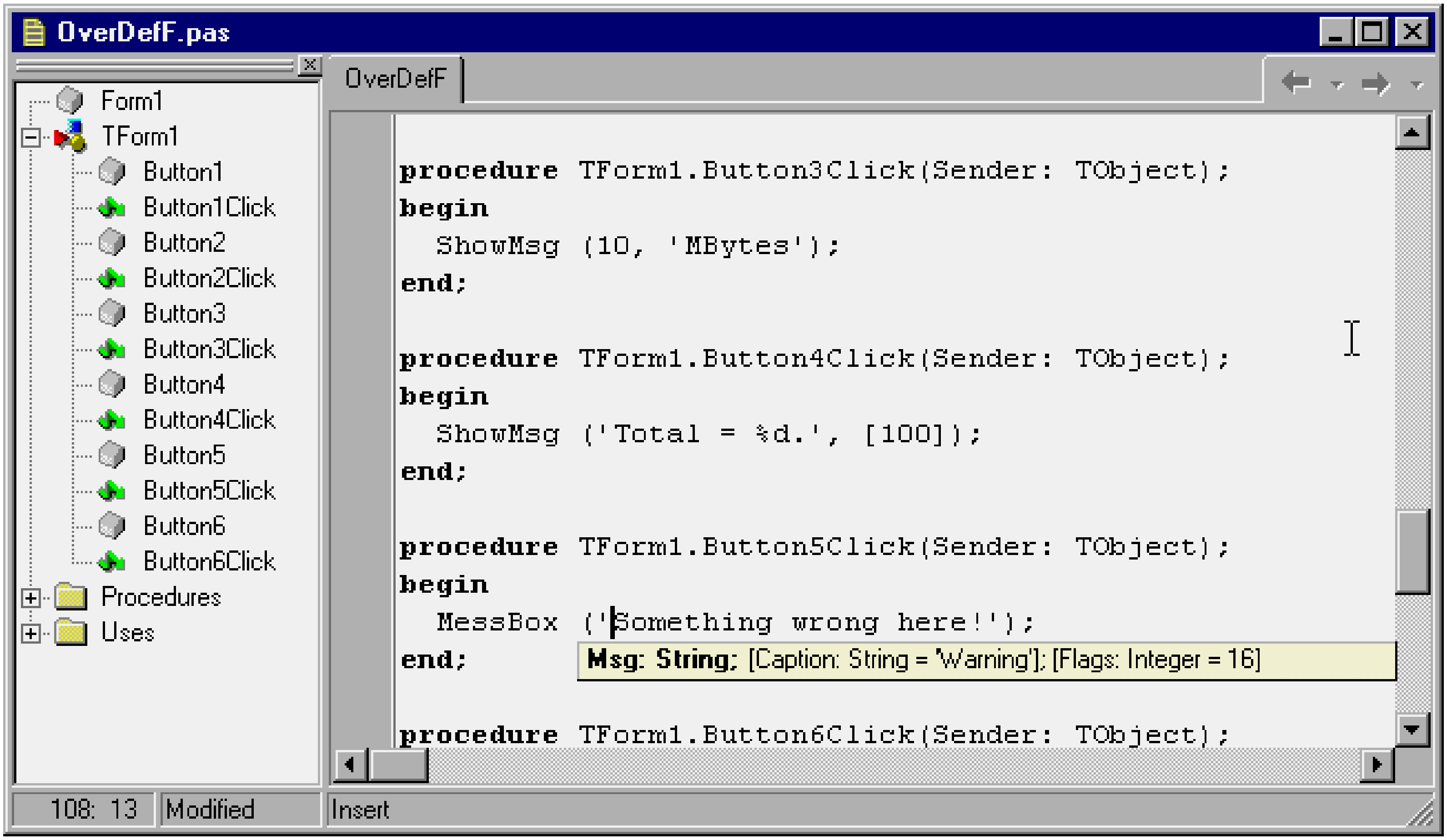
```
procedure MessBox (Msg: string;
  Caption: string = 'Warning';
  Flags: LongInt = mb_OK or mb_IconHand);
begin
  Application.MessageBox (PChar (Msg),
    PChar (Caption), Flags);
end;
```

With this definition, we can call the procedure in each of the following ways:

```
MessBox ('Something wrong here!');
MessBox ('Something wrong here!', 'Attention');
MessBox ('Hello', 'Message', mb_OK);
```

In Figure 6.5 you can see that Delphi's Code Parameters properly use a different style to indicate the parameters that have a default value, so you can easily determine which parameters can be omitted.

Figure 6.5: Delphi's Code Parameters mark out with square brackets the parameters that have default values; you can omit these in the call.



```
procedure TForm1.Button3Click(Sender: TObject);
begin
  ShowMsg (10, 'MBytes');
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  ShowMsg ('Total = %d.', [100]);
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  MessBox ('Something wrong here!');
end;

procedure TForm1.Button6Click(Sender: TObject);
```

Msg: String; [Caption: String = 'Warning']; [Flags: Integer = 16]

Notice that Delphi doesn't generate any special code to support default parameters; nor does it create multiple copies of the routines. The missing parameters are simply added by the compiler to the calling code.

There is one important restriction affecting the use of default parameters: You cannot "skip" parameters. For example, you can't pass the third parameter to the function after omitting the second one:

```
MessBox ('Hello', mb_OK); // error
```

This is the main rule for default parameters: In a call, you can only omit parameters starting from the last one. In other words, if you omit a parameter you must omit also the following ones.

There are a few other rules for default parameters as well:

- Parameters with default values must be at the end of the parameters list.
- Default values must be constants. Obviously, this limits the types you can use with default parameters. For example, a dynamic array or an interface type cannot have a default parameter other than nil; records cannot be used at all.
- Default parameters must be passed by value or as const. A reference (var) parameter cannot have a default value.

Using default parameters and overloading at the same time can cause quite a few problems, as the two features might conflict. For example, if I add to the previous example the following new version of the ShowMsg procedure:

```
procedure ShowMsg (Str: string; I: Integer = 0); overload;
begin
  MessageDlg (Str + ': ' + IntToStr (I),
    mtInformation, [mbOK], 0);
end;
```

then the compiler won't complain-this is a legal definition. However, the call:

```
ShowMsg ('Hello');
```

is flagged by the compiler as Ambiguous overloaded call to 'ShowMsg'. Notice that this error shows up in a line of code that compiled correctly before the new overloaded definition. In practice, we have no way to call the ShowMsg procedure with one string parameter, as the compiler doesn't know whether we want to call the version with only the string parameter or the one with the string parameter and the integer parameter with a default value. When it has a similar doubt, the compiler stops and asks the programmer to state his or her intentions more clearly.

Conclusion

Writing procedure and functions is a key element of programming, although in Delphi you'll tend to write methods -- procedures and functions connected with classes and objects.

Instead of moving on to object-oriented features, however, the next few chapters give you some details on other Pascal programming elements, starting with strings.

Next Chapter: Handling Strings

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 7 Handling Strings

String handling in Delphi is quite simple, but behind the scenes the situation is quite complex. Pascal has a traditional way of handling strings, Windows has its own way, borrowed from the C language, and 32-bit versions of Delphi include a powerful long string data type, which is the default string type in Delphi.

Types of Strings

In Borland's Turbo Pascal and in 16-bit Delphi, the typical string type is a sequence of characters with a length byte at the beginning, indicating the current size of the string. Because the length is expressed by a single byte, it cannot exceed 255 characters, a very low value that creates many problems for string manipulation. Each string is defined with a fixed size (which by default is the maximum, 255), although you can declare shorter strings to save memory space.

A string type is similar to an array type. In fact, a string is almost an array of characters. This is demonstrated by the fact that you can access a specific string character using the [] notation.

To overcome the limits of traditional Pascal strings, the 32-bit versions of Delphi support long strings. There are actually three string types:

- The ShortString type corresponds to the typical Pascal strings, as described before. These strings have a limit of 255 characters and correspond to the strings in the 16-bit version of Delphi. Each element of a short string is of type ANSIChar (the standard character type).
- The ANSIString type corresponds to the new variable-length long strings. These strings are allocated dynamically, are reference counted, and use a copy-on-write technique. The size of these strings is almost unlimited (they can store up to two billion characters!). They are also based on the ANSIChar type.
- The WideString type is similar to the ANSIString type but is based on the WideChar type-it stores Unicode characters.

Using Long Strings

If you simply use the string data type, you get either short strings or ANSI strings, depending on the value of the $H compiler directive. $H+ (the default) stands for long strings (the ANSIString type), which is what is used by the components of the Delphi library.

Delphi long strings are based on a reference-counting mechanism, which keeps track of how many string variables are referring to the same string in memory. This reference-counting is used also to free the memory when a string isn't used anymore-that is, when the reference count reaches zero.

If you want to increase the size of a string in memory but there is something else in the adjacent memory, then the string cannot grow in the same memory location, and a full copy of the string must therefore be made in another location. When this situation occurs, Delphi's run-time support reallocates the string for you in a completely transparent way. You simply set the maximum size of the string with the SetLength procedure, effectively allocating the required amount of memory:

```
SetLength (String1, 200);
```

The SetLength procedure performs a memory request, not an actual memory allocation. It reserves the required memory space for future use, without actually using the memory. This technique is based on a feature of the Windows operating systems and is used by Delphi for all dynamic memory allocations. For example, when you request a very large array, its memory is reserved but not allocated.

Setting the length of a string is seldom necessary. The only case in which you must allocate memory for the long string using SetLength is when you have to pass the string as a parameter to an API function (after the proper typecast), as I'll show you shortly.

Looking at Strings in Memory

To help you better understand the details of memory management for strings, I've written the simple StrRef example. In this program I declare two global strings: Str1 and Str2. When the first of the two buttons is pressed, the program assigns a constant string to the first of the two variables and then assigns the second variable to the first:

```
Str1 := 'Hello';
Str2 := Str1;
```

Besides working on the strings, the program shows their internal status in a list box, using the following StringStatus function:

```
function StringStatus (const Str: string): string;
begin
  Result := 'Address: ' + IntToStr (Integer (Str)) +
    ', Length: ' + IntToStr (Length (Str)) +
    ', References: ' + IntToStr (PInteger (Integer (Str) - 8)^) +
    ', Value: ' + Str;
end;
```

It is vital in the StringStatus function to pass the string parameter as a const parameter. Passing this parameter by copying will cause the side effect of having one extra reference to the string while the function is being executed. By contrast, passing the parameter via a reference (var) or constant (const) parameter doesn't imply a further reference to the string. In this case I've used a const parameter, as the function is not supposed to modify the string.

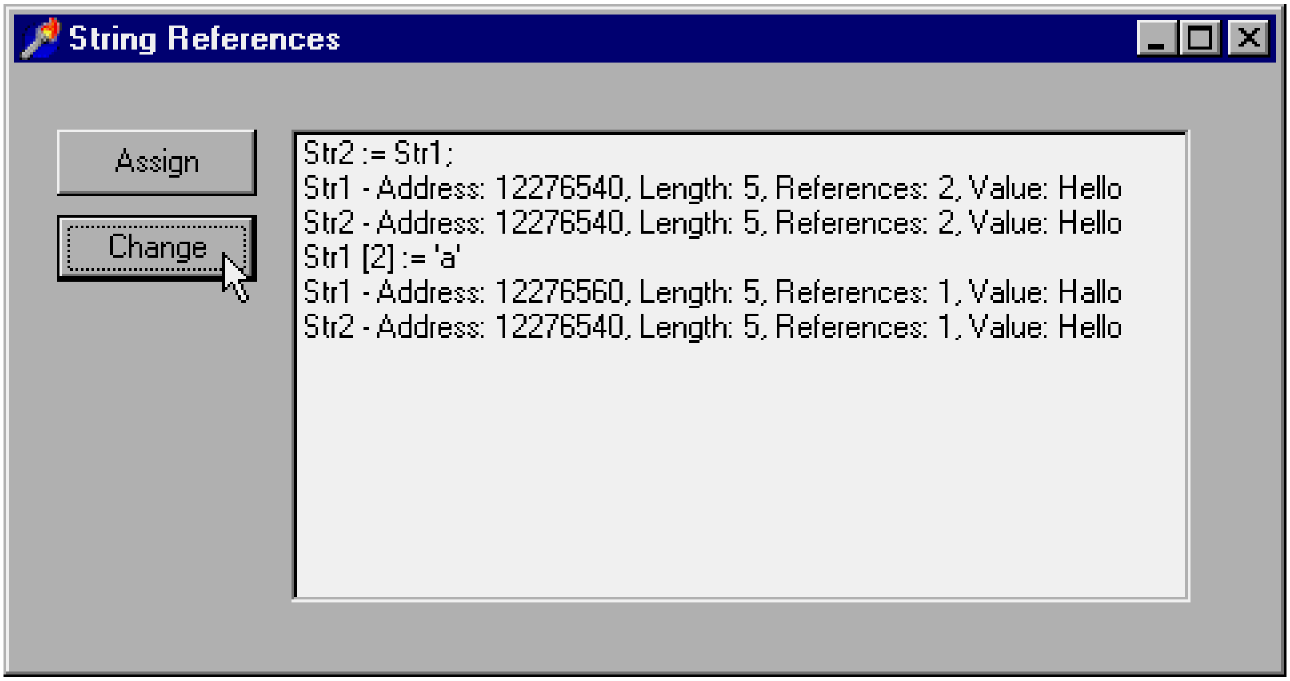
To obtain the memory address of the string (useful to determine its actual identity and to see when two different strings refer to the same memory area), I've simply made a hard-coded typecast from the string type to the Integer type. Strings are references-in practice, they're pointers: Their value holds the actual memory location of the string.

To extract the reference count, I've based the code on the little-known fact that the length and reference count are actually stored in the string, before the actual text and before the position the string variable points to. The (negative) offset is -4 for the length of the string (a value you can extract more easily using the Length function) and -8 for the reference count.

Keep in mind that this internal information about offsets might change in future versions of Delphi; there is also no guarantee that similar undocumented features will be maintained in the future.

By running this example, you should get two strings with the same content, the same memory location, and a reference count of 2, as shown in the upper part of the list box of Figure 2.1. Now if you change the value of one of the two strings (it doesn't matter which one), the memory location of the updated string will change. This is the effect of the copy-on-write technique.

Figure 7.1: The StrRef example shows the internal status of two strings, including the current reference count.



```
Str2 := Str1;
Str1 - Address: 12276540, Length: 5, References: 2, Value: Hello
Str2 - Address: 12276540, Length: 5, References: 2, Value: Hello
Str1 [2] := 'a'
Str1 - Address: 12276560, Length: 5, References: 1, Value: Hallo
Str2 - Address: 12276540, Length: 5, References: 1, Value: Hello
```

We can actually produce this effect, shown in the second part of the list box of Figure 7.1, by writing the following code for the OnClick event handler of the second button:

```
procedure TFormStrRef.BtnChangeClick(Sender: TObject);
begin
  Str1 [2] := 'a';
  ListBox1.Items.Add ('Str1 [2] := ''a''');
  ListBox1.Items.Add ('Str1 - ' + StringStatus (Str1));
  ListBox1.Items.Add ('Str2 - ' + StringStatus (Str2));
end;
```

Notice that the code of the BtnChangeClick method can be executed only after the BtnAssignClick method. To enforce this, the program starts with the second button disabled (its Enabled property is set to False); it enables the button at the end of the first method. You can freely extend this example and use the StringStatus function to explore the behavior of long strings in many other circumstances.

Delphi Strings and Windows PChars

Another important point in favor of using long strings is that they are null-terminated. This means that they are fully compatible with the C language null-terminated strings used by Windows. A null-terminated string is a sequence of characters followed by a byte that is set to zero (or null). This can be expressed in Delphi using a zero-based array of characters, the data type typically used to implement strings in the C language. This is the reason null-terminated character arrays are so common in the Windows API functions (which are based on the C language). Since Pascal's long strings are fully compatible with C null-terminated strings, you can simply use long strings and cast them to PChar when you need to pass a string to a Windows API function.

For example, to copy the caption of a form into a PChar string (using the API function GetWindowText) and then copy it into the Caption of the button, you can write the following code:

```
procedure TForm1.Button1Click (Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  Button1.Caption := S1;
end;
```

You can find this code in the LongStr example. Note that if you write this code but fail to allocate the memory for the string with SetLength, the program will probably crash. If you are using a PChar to pass a value (and not to receive one as in the code above), the code is even simpler, because there is no need to define a temporary string and initialize it. The following line of code passes the Caption property of a label as a parameter to an API function, simply by typecasting it to PChar:

```
SetWindowText (Handle, PChar (Label1.Caption));
```

When you need to cast a WideString to a Windows-compatible type, you have to use PWideChar instead of PChar for the conversion. Wide strings are often used for OLE and COM programs.

Having presented the nice picture, now I want to focus on the pitfalls. There are some problems that might arise when you convert a long string into a PChar. Essentially, the underlying problem is that after this conversion, you become responsible for the string and its contents, and Delphi won't help you anymore. Consider the following limited change to the first program code fragment above, Button1Click:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := S1 + ' is the title'; // this won't work
  Button1.Caption := S1;
end;
```

This program compiles, but when you run it, you are in for a surprise: The Caption of the button will have the original text of the window title, without the text of the constant string you have added to it. The problem is that when Windows writes to the string (within the GetWindowText API call), it doesn't set the length of the long Pascal string properly. Delphi still can use this string for output and can figure out when it ends by looking for the null terminator, but if you append further characters after the null terminator, they will be skipped altogether.

How can we fix this problem? The solution is to tell the system to convert the string returned by the GetWindowText API call back to a Pascal string. However, if you write the following code:

```
S1 := String (S1);
```

the system will ignore it, because converting a data type back into itself is a useless operation. To obtain the proper long Pascal string, you need to recast the string to a PChar and let Delphi convert it back again properly to a string:

```
S1 := String (PChar (S1));
```

Actually, you can skip the string conversion, because PChar-to-string conversions are automatic in Delphi. Here is the final code:

```
procedure TForm1.Button3Click(Sender: TObject);
```

```
var
  S1: String;
begin
  SetLength (S1, 100);
  GetWindowText (Handle, PChar (S1), Length (S1));
  S1 := String (PChar (S1));
  S1 := S1 + ' is the title';
  Button3.Caption := S1;
end;
```

An alternative is to reset the length of the Delphi string, using the length of the PChar string, by writing:

```
SetLength (S1, StrLen (PChar (S1)));
```

You can find three versions of this code in the LongStr example, which has three buttons to execute them. However, if you just need to access the title of a form, you can simply use the Caption property of the form object itself. There is no need to write all this confusing code, which was intended only to demonstrate the string conversion problems. There are practical cases when you need to call Windows API functions, and then you have to consider this complex situation.

Formatting Strings

Using the plus (+) operator and some of the conversion functions (such as IntToStr) you can indeed build complex strings out of existing values. However, there is a different approach to formatting numbers, currency values, and other strings into a final string. You can use the powerful Format function or one of its companion functions.

The Format function requires as parameters a string with the basic text and some placeholders (usually marked by the % symbol) and an array of values, one for each placeholder. For example, to format two numbers into a string you can write:

```
Format ('First %d, Second %d', [n1, n2]);
```

where n1 and n2 are two Integer values. The first placeholder is replaced by the first value, the second matches the second, and so on. If the output type of the placeholder (indicated by the letter after the % symbol) doesn't match the type of the corresponding parameter, a runtime error occurs. Having no compile-time type checking is actually the biggest drawback of using the Format function.

The Format function uses an open-array parameter (a parameter that can have an arbitrary number of values), something I'll discuss toward the end of this chapter. For the moment, though, notice only the array-like syntax of the list of values passed as the second parameter.

Besides using %d, you can use one of many other placeholders defined by this function and briefly listed in Table 7.1. These placeholders provide a default output for the given data type. However, you can use further format specifiers to alter the default output. A width specifier, for example, determines a fixed number of characters in the output, while a precision specifier indicates the number of decimal digits. For example,

```
Format ('%8d', [n1]);
```

converts the number n1 into an eight-character string, right-aligning the text (use the minus (-) symbol to specify left-justification) filling it with white spaces.

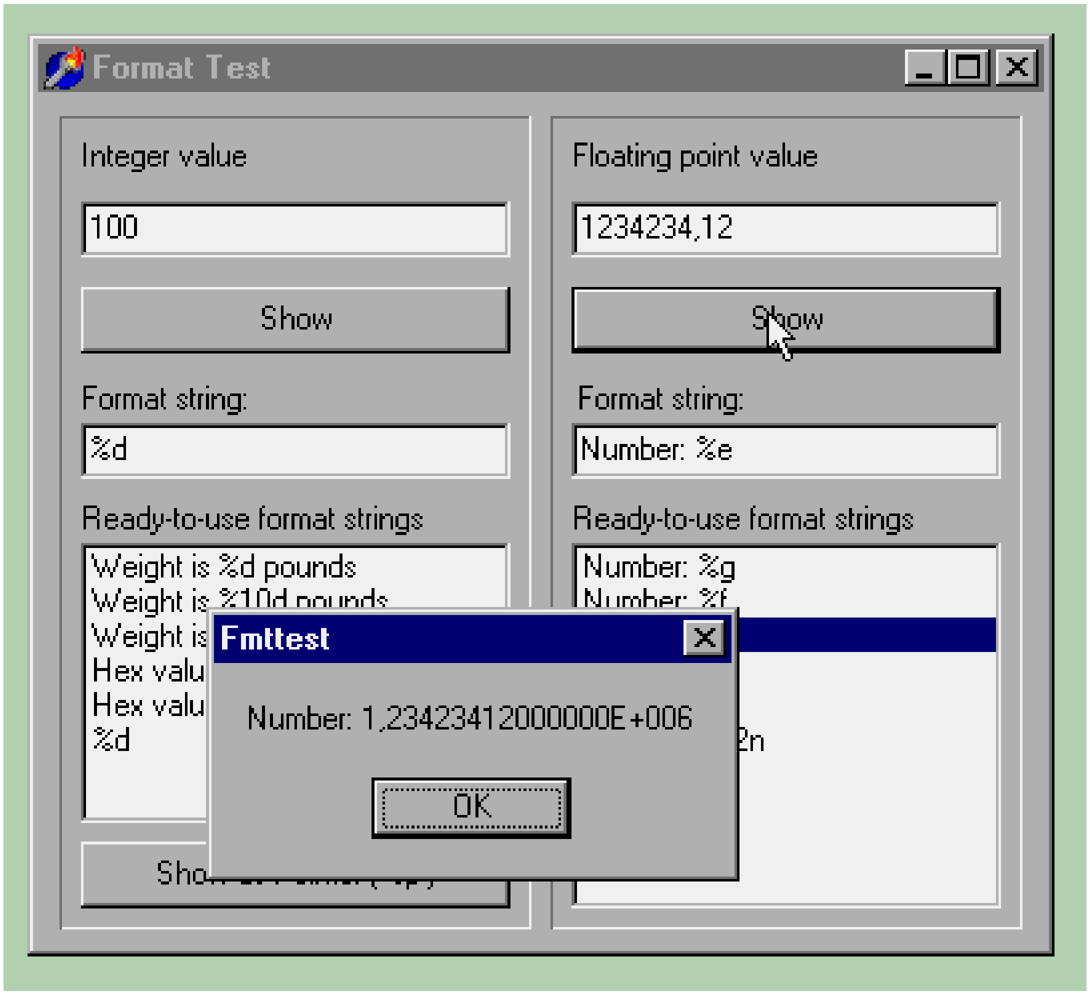
Table 7.1: Type Specifiers for the Format Function

TYPE SPECIFIER	DESCRIPTION
d (decimal)	The corresponding integer value is converted to a string of decimal digits.
x (hexadecimal)	The corresponding integer value is converted to a string of hexadecimal digits.
p (pointer)	The corresponding pointer value is converted to a string expressed with hexadecimal digits.
s (string)	The corresponding string, character, or PChar value is copied to the output string.
e (exponential)	The corresponding floating-point value is converted to a string based on exponential notation.
f (floating point)	The corresponding floating-point value is converted to a string based on floating point notation.
g (general)	The corresponding floating-point value is converted to the shortest possible decimal string using either floating-point or exponential notation.
n (number)	The corresponding floating-point value is converted to a floating-point string but also uses thousands separators.
m (money)	The corresponding floating-point value is converted to a string representing a currency amount. The conversion is based on regional settings-see the Delphi Help file under Currency and date/time formatting variables.

The best way to see examples of these conversions is to experiment with format strings yourself. To make this easier I've written the FmtTest program, which allows a user to provide formatting strings for integer and floating-point numbers. As you can see in Figure 7.2, this program displays a form divided into two parts. The left part is for Integer numbers, the right part for floating-point numbers.

Each part has a first edit box with the numeric value you want to format to a string. Below the first edit box there is a button to perform the formatting operation and show the result in a message box. Then comes another edit box, where you can type a format string. As an alternative you can simply click on one of the lines of the ListBox component, below, to select a predefined formatting string. Every time you type a new formatting string, it is added to the corresponding list box (note that by closing the program you lose these new items).

Figure 7.2: The output of a floating-point value from the FmtTest program



100

%d

Weight is %d pounds
Weight is %10d pounds
Weight is
Hex valu
Hex valu
%d

1234234,12

Number: %e

Number: %g
Number: %f
2n

Number: 1,23423412000000E+006

The code of this example simply uses the text of the various controls to produce its output. This is one of the three methods connected with the Show buttons:

```
procedure TFormFmtTest.BtnIntClick(Sender: TObject);
begin
  ShowMessage (Format (EditFmtInt.Text,
    [StrToInt (EditInt.Text)]));
  // if the item is not there, add it
  if ListBoxInt.Items.IndexOf (EditFmtInt.Text) < 0 then
    ListBoxInt.Items.Add (EditFmtInt.Text);
end;
```

The code basically does the formatting operation using the text of the EditFmtInt edit box and the value of the EditInt control. If the format string is not already in the list box, it is then added to it. If the user instead clicks on an item in the list box, the code moves that value to the edit box:

```
procedure TFormFmtTest.ListBoxIntClick(Sender: TObject);
begin
  EditFmtInt.Text := ListBoxInt.Items [
    ListBoxInt.ItemIndex];
end;
```

Conclusion

Strings a certainly a very common data type. Although you can safely use them in most cases without understanding how they work, this chapter should have made clear the exact behavior of strings, making it possible for you to use all the power of this data type.

Strings are handled in memory in a special dynamic way, as happens with dynamic arrays. This is the topic of the next chapter.

Next Chapter: Memory

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 8 Memory

Author's Note: This chapter will cover memory handling, discuss the various memory areas, and introduce dynamic arrays. Temporarily only this last part is available.

Delphi 4 Dynamic Arrays

Traditionally, the Pascal language has always had fixed-size arrays. When you declare a data type using the array construct, you have to specify the number of elements of the array. As expert programmers probably know, there were a few techniques you could use to implement dynamic arrays, typically using pointers and manually allocating and freeing the required memory.

Delphi 4 introduces a very simple implementation of dynamic arrays, modeling them after the dynamic long string type I've just covered. As long strings, dynamic arrays are dynamically allocated and reference counted, but they do not offer a copy-on-write technique. That's not a big problem, as you can deallocate an array by setting its variable to nil.

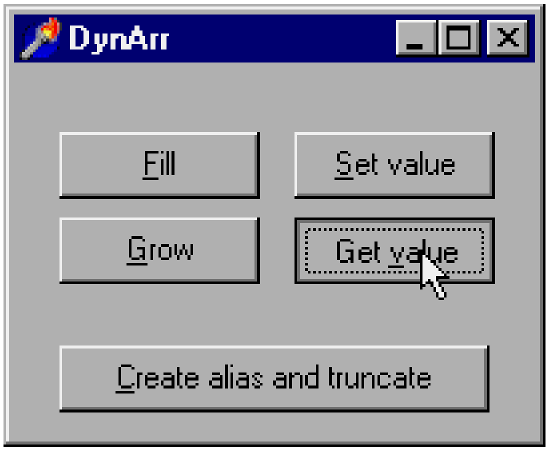
You can now simply declare an array without specifying the number of elements and then allocate it with a given size using the SetLength procedure. The same procedure can also be used to resize an array without losing its content. There are also other string-oriented procedures, such as the Copy function, that you can use on arrays.

Here is a small code excerpt, underscoring the fact that you must both declare and allocate memory for the array before you can start using it:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Array1: array of Integer;
begin
  Array1 [1] := 100; // error
  SetLength (Array1, 100);
  Array1 [99] := 100; // OK
  ...
end;
```

As you indicate only the number of elements of the array, the index invariably starts from 0. Generic arrays in Pascal account for a non-zero low bound and for non-integer indexes, two features that dynamic arrays don't support. To learn the status of a dynamic array, you can use the Length, High, and Low functions, as with any other array. For dynamic arrays, however, Low always returns 0, and High always returns the length minus one. This implies that for an empty array High returns -1 (which, when you think about it, is a strange value, as it is lower than that returned by Low).

Figure 8.1: The form of the DynArr example



After this short introduction I can show you a simple example, called DynArr and shown in Figure 8.1. It is indeed simple because there is nothing very complex about dynamic arrays. I'll also use it to show a few possible errors programmers might make. The program declares two global arrays and initializes the first in the OnCreate handler:

```
var
  Array1, Array2: array of Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  // allocate
  SetLength (Array1, 100);
end;
```

This sets all the values to zero. This initialization code makes it possible to start reading and writing values of the array right away, without any fear of memory errors. (Assuming, of course, that you don't try to access items beyond the upper bound of the array.) For an even better initialization, the program has a button that writes into each cell of the array:

```
procedure TForm1.btnFillClick(Sender: TObject);
var
  I: Integer;
begin
  for I := Low (Array1) to High (Array1) do
    Array1 [I] := I;
end;
```

The Grow button allows you to modify the size of the array without losing its contents. You can test this by using the Get value button after pressing the Grow button:

```
procedure TForm1.btnGrowClick(Sender: TObject);
begin
  // grow keeping existing values
  SetLength (Array1, 200);
end;

procedure TForm1.btnGetClick(Sender: TObject);
begin
  // extract
  Caption := IntToStr (Array1 [99]);
end;
```

The only slightly complex code is in the OnClick event of the Alias button. The program copies one array to the other one with the := operator, effectively creating an alias (a new variable referring to the same array in memory). At this point, however, if you modify one of the arrays, the other is affected as well, as they both refer to the same memory area:

```
procedure TForm1.btnAliasClick(Sender: TObject);
begin
  // alias
  Array2 := Array1;
  // change one (both change)
  Array2 [99] := 1000;
  // show the other
  Caption := IntToStr (Array1 [99]);
```

The btnAliasClick method does two more operations. The first is an equality test on the arrays. This tests not the actual elements of the structures but rather the memory areas the arrays refer to, checking whether the variables are two aliases of the same array in memory:

```
procedure TForm1.btnAliasClick(Sender: TObject);
begin
  ...
  if Array1 = Array2 then
    Beep;
  // truncate first array
  Array1 := Copy (Array2, 0, 10);
end;
```

The second is a call to the Copy function, which not only moves data from one array to the other, but also replaces the first array with a new one created by the function. The effect is that the Array1 variable now refers to an array of 11 elements, so that pressing the Get value or Set value buttons produces a memory error and raises an exception (unless you have range-checking turned off, in which case the error remains but the exception is not displayed). The code of the Fill button continues to work fine even after this change, as the items of the array to modify are determined using its current bounds.

Conclusion

This chapter temporarily covers only dynamic arrays, certainly an important element for memory management, but only a portion of the entire picture. More material will follow.

The memory structure described in this chapter is typical of Windows programming, a topic I'll introduce in the next chapter (without going to the full extent of using the VCL, though).

Next Chapter: Windows Programming

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 9: Windows Programming

Delphi provides a complete encapsulation of the low-level Windows API using Object Pascal and the Visual Component Library (VCL), so it is rarely necessary to build Windows applications using plain Pascal and calling Windows API functions directly. Nonetheless, programmers who want to use some special techniques not supported by the VCL still have that option in Delphi. You would only want to take this approach for very special cases, such as the development of new Delphi components based on unusual API calls, and I don't want to cover the details. Instead, we'll look at a few elements of Delphi's interaction with the operating system and a couple of techniques that Delphi programmers can benefit from.

Windows Handles

Among the data types introduced by Windows in Delphi, handles represent the most important group. The name of this data type is *THandle*, and the type is defined in the Windows unit as:

```
type
  THandle = LongWord;
```

Handle data types are implemented as numbers, but they are not used as such. In Windows, a handle is a reference to an internal data structure of the system. For example, when you work with a window (or a Delphi form), the system gives you a handle to the window. The system informs you that the window you are working with is window number 142, for example. From that point on, your application can ask the system to operate on window number 142—moving it, resizing it, reducing it to an icon, and so on. Many Windows API functions, in fact, have a handle as the first parameter. This doesn't apply only to functions operating on windows; other Windows API functions have as their first parameter a GDI handle, a menu handle, an instance handle, a bitmap handle, or one of the many other handle types.

In other words, a handle is an internal code you can use to refer to a specific element handled by the system, including a window, a bitmap, an icon, a memory block, a cursor, a font, a menu, and so on. In Delphi, you seldom need to use handles directly, since they are hidden inside forms, bitmaps, and other Delphi objects. They become useful when you want to call a Windows API function that is not supported by Delphi.

To complete this description, here is a simple example demonstrating Windows handles. The WHandle program has a simple form, containing just a button. In the code, I respond to the OnCreate event of the form and the OnClick event of the button, as indicated by the following textual definition of the main form:

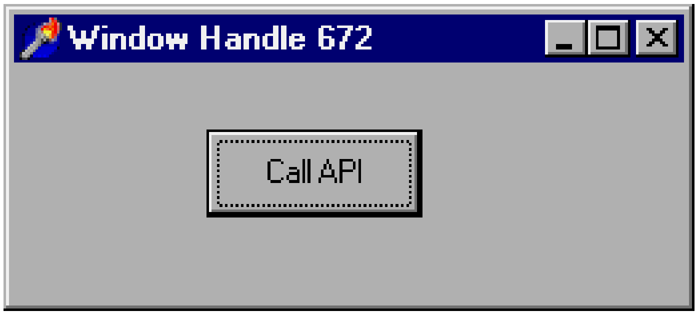
```
object FormWHandle: TFormWHandle
  Caption = 'Window Handle'
  OnCreate = FormCreate
  object BtnCallAPI: TButton
    Caption = 'Call API'
    OnClick = BtnCallAPIClick
  end
end
```

As soon as the form is created, the program retrieves the handle of the window corresponding to the form, by accessing the Handle property of the form itself. We call IntToStr to convert the numeric value of the handle into a string, and we append that to the caption of the form, as you can see in Figure 9.1:

```
procedure TFormWHandle.FormCreate(Sender: TObject);
begin
  Caption := Caption + ' ' + IntToStr (Handle);
end;
```

Because FormCreate is a method of the form's class, it can access other properties and methods of the same class directly. Therefore, in this procedure we can simply refer to the Caption of the form and its Handle property directly.

Figure 9.1: The WHandle example shows the handle of the form window. Every time you run this program you'll get a different value.



If you run this program several times you'll generally get different values for the handle. This value, in fact, is determined by Windows and is sent back to the application. (Handles are never determined by the program, and they have no predefined values; they are determined by the system, which generates new values each time you run a program.)

When the user presses the button, the program simply calls a Windows API function, SetWindowText, which changes the text or caption of the window passed as the first parameter. To be more precise, the first parameter of this API function is the handle of the window we want to modify:

```
procedure TFormWHandle.BtnCallAPIClick(Sender: TObject);
begin
  SetWindowText (Handle, 'Hi');
end;
```

This code has the same effect as the previous event handler, which changed the text of the window by giving a new value to the Caption property of the form. In this case calling an API function makes no sense, because there is a simpler Delphi technique. Some API functions, however, have no correspondence in Delphi, as we'll see in more advanced examples later in the book.

External Declarations

Another important element for Windows programming is represented by external declarations. Originally used to link the Pascal code to external functions that were written in assembly language, the external declaration is used in Windows programming to call a function from a DLL (a dynamic link library). In Delphi, there are a number of such declarations in the Windows unit:

```
// forward declaration
function LineTo (DC: HDC; X, Y: Integer): BOOL; stdcall;

// external declaration (instead of actual code)
function LineTo; external 'gdi32.dll' name 'LineTo';
```

This declaration means that the code of the function LineTo is stored in the GDI32.DLL dynamic library (one of the most important Windows system libraries) with the same name we are using in our code. Inside an external declaration, in fact, we can specify that our function refer to a function of a DLL that originally had a different name.

You seldom need to write declarations like the one just illustrated, since they are already listed in the Windows unit and many other Delphi system units. The only reason you might need to write this external declaration code is to call functions from a custom DLL, or to call undocumented Windows functions.

Note: In the 16-bit version of Delphi, the external declaration used the name of the library without the extension, and was followed by the name directive (as in the code above) or by an alternative index directive, followed by the ordinal number of the function inside the DLL. The change reflects a system change in the way libraries are accessed: Although Win32 still allows access to DLL functions by number, Microsoft has stated this won't be supported in the future. Notice also that the Windows unit replaces the WinProcs and WinTypes units of the 16-bit version of Delphi.

A Windows Callback Function

We've seen in Chapter 6 that Objet Pascal supports procedural types. A common use of procedural types is to provide callback functions to a Windows API function.

First of all, what is a callback function? The idea is that some API function performs a given action over a number of internal elements of the system, such as all of the windows of a certain kind. Such a function, also called an enumerated function, requires as a parameter the action to be performed on each of the elements, which is passed as a function or procedure compatible with a given procedural type. Windows uses callback functions in other circumstances, but we'll limit our study to this simple case.

Now consider the EnumWindows API function, which has the following prototype (copied from the Win32 Help file):

```
BOOL EnumWindows(
  WNDENUMPROC lpEnumFunc,  // address of callback function
  LPARAM lParam // application-defined value
  );
```

Of course, this is the C language definition. We can look inside the Windows unit to retrieve the corresponding Pascal language definition:

```
function EnumWindows (
  lpEnumFunc: TFNWndEnumProc;
  lParam: LPARAM): BOOL; stdcall;
```

Consulting the help file, we find that the function passed as a parameter should be of the following type (again in C):

```
BOOL CALLBACK EnumWindowsProc (
  HWND hwnd, // handle of parent window
  LPARAM lParam // application-defined value
  );
```

This corresponds to the following Delphi procedural type definition:

```
type
  EnumWindowsProc = function (Hwnd: THandle;
    Param: Pointer): Boolean; stdcall;
```

The first parameter is the handle of each main window in turn, while the second is the value we've passed when calling the EnumWindows function. Actually in Pascal the TFNWndEnumProc type is not properly defined; it is simply a pointer. This means we need to provide a function with the proper parameters and then use it as a pointer, taking the address of the function instead of calling it. Unfortunately, this also means that the compiler will provide no help in case of an error in the type of one of the parameters.

Windows requires programmers to follow the stdcall calling convention every time we call a Windows API function or pass a callback function to the system. Delphi, by default, uses a different and more efficient calling convention, indicated by the register keyword.

Here is the definition of a proper compatible function, which reads the title of the window into a string, then adds it to a ListBox of a given form:

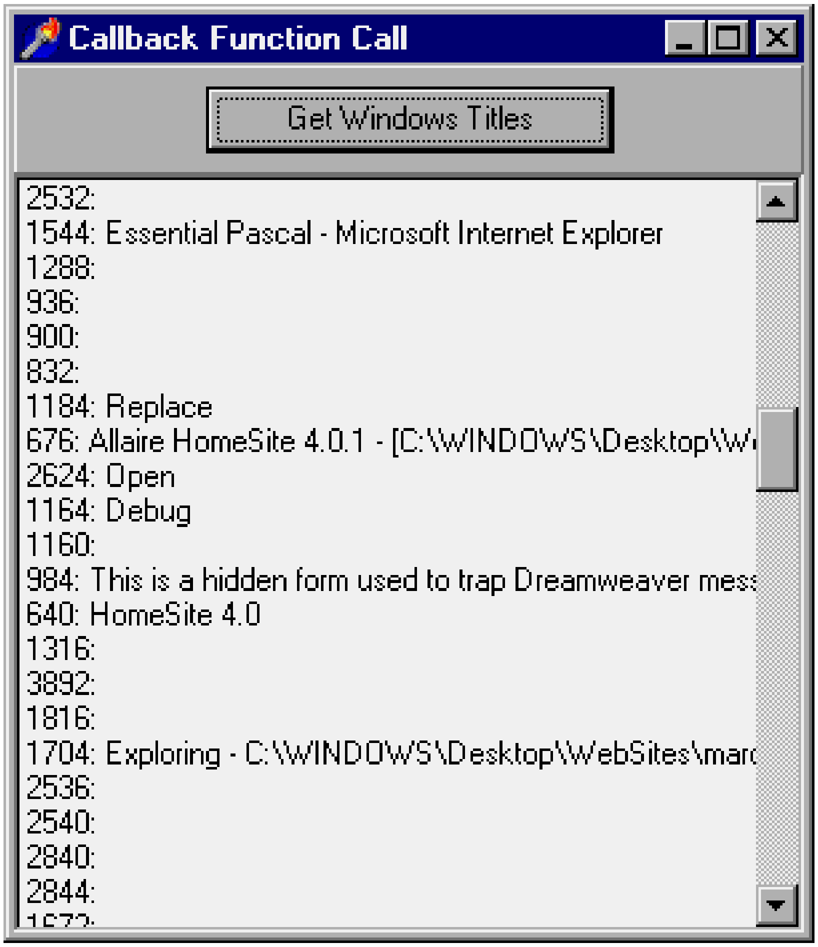
```
function GetTitle (Hwnd: THandle; Param: Pointer): Boolean; stdcall;
var
  Text: string;
begin
  SetLength (Text, 100);
  GetWindowText (Hwnd, PChar (Text), 100);
  FormCallBack.ListBox1.Items.Add (
    IntToStr (Hwnd) + ': ' + Text);
  Result := True;
end;
```

The form has a ListBox covering almost its whole area, along with a small panel on the top hosting a button. When the button is pressed, the EnumWindows API function is called, and the GetTitle function is passed as its parameter:

```
procedure TFormCallback.BtnTitlesClick(Sender: TObject);
var
  EWProc: EnumWindowsProc;
begin
  ListBox1.Items.Clear;
  EWProc := GetTitle;
  EnumWindows (@EWProc, 0);
end;
```

I could have called the function without storing the value in a temporary procedural type variable first, but I wanted to make clear what is going on in this example. The effect of this program is actually quite interesting, as you can see in Figure 9.2. The Callback example shows a list of all the existing main windows running in the system. Most of them are hidden windows you usually never see (and many actually have no caption).

Figure 9.2: The output of the Callback example, listing the current main windows (visible and hidden).



```
2532:
1544: Essential Pascal - Microsoft Internet Explorer
1288:
936:
900:
832:
1184: Replace
676: Allaire HomeSite 4.0.1 - [C:\WINDOWS\Desktop\W
2624: Open
1164: Debug
1160:
984: This is a hidden form used to trap Dreamweaver mess
640: HomeSite 4.0
1316:
3892:
1816:
1704: Exploring - C:\WINDOWS\Desktop\WebSites\marc
2536:
2540:
2840:
2844:
```

A Minimal Windows Program

To complete the coverage of Windows programming and the Pascal language, I want to show you a very simple but complete application built without using the VCL. The program simply takes the command-line parameter (stored by the system in the cmdLine global variable) and then extracts information from it with the ParamCount and ParamStr Pascal functions. The first of these functions returns the number of parameters; the second returns the parameter in a given position.

Although users seldom specify command-line parameters in a graphical user interface environment, the Windows command-line parameters are important to the system. For example, once you have defined an association between a file extension and an application, you can simply run a program by selecting an associated file. In practice, when you double-click on a file, Windows starts the associated program and passes the selected file as a command-line parameter.

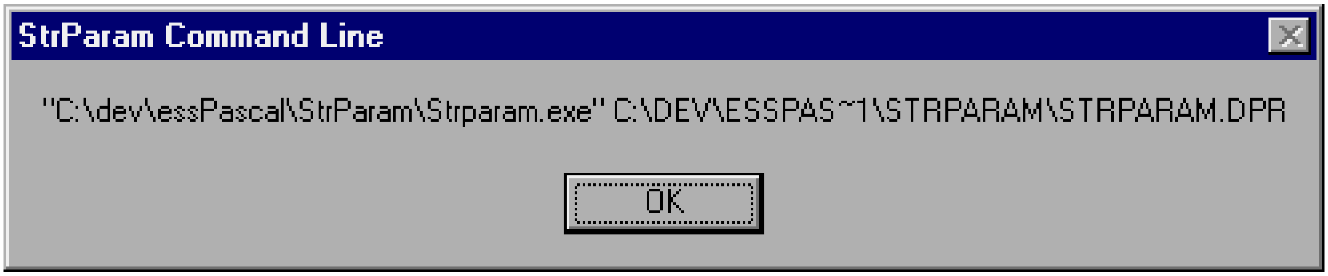
Here is the complete source code of the project (a DPR file, not a PAS file):

```
program Strparam;

uses
  Windows;

begin
  // show the full string
  MessageBox (0, cmdLine,
    'StrParam Command Line', MB_OK);

  // show the first parameter
  if ParamCount > 0 then
    MessageBox (0, PChar (ParamStr (1)),
      '1st StrParam Parameter', MB_OK)
  else
    MessageBox (0, PChar ('No parameters'),
      '1st StrParam Parameter', MB_OK);
end.
```

The output code uses the MessageBox API function, simply to avoid getting the entire VCL into the project. A pure Windows program as the one above, in fact, has the advantage of a very small memory footprint: The executable file of the program is about 16 Kbytes.

To provide a command-line parameter to this program, you can use Delphi's Run > Parameters menu command. Another technique is to open the Windows Explorer, locate the directory that contains the executable file of the program, and drag the file you want to run onto the executable file. The Windows Explorer will start the program using the name of the dropped file as a command-line parameter. Figure 9.3 shows both the Explorer and the corresponding output.

Figure 9.3: You can provide a command-line parameter to the StrParam example by dropping a file over the executable file in the Windows Explorer.


StrParam Command Line

"C:\dev\essPascal\StrParam\Strparam.exe" C:\DEV\ESSPAS~1\STRPARAM\STRPARAM.DPR

Conclusion

In this chapter we've seen a low-level introduction to Windows programming, discussing handles and a very simple Windows program. For normal Windows programming tasks, you'll generally use the visual development support provided by Delphi and based on the VCL. But this is beyond the scope of this book, which is the Pascal language.

Next chapter covers variants, a very strange addition to Pascal type system, introduced to provide full OLE support.

Next Chapter: Variants

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 10 Variants

To provide full OLE support, the 32-bit version of Delphi includes the Variant data type. Here I want to discuss this data type from a general perspective. The Variant type, in fact, has a pervasive effect on the whole language, and the Delphi components library also uses them in ways not related to OLE programming.

Variants Have No Type

In general, you can use variants to store any data type and perform numerous operations and type conversions. Notice that this goes against the general approach of the Pascal language and against good programming practices. A variant is type-checked and computed at run time. The compiler won't warn you of possible errors in the code, which can be caught only with extensive testing. On the whole, you can consider the code portions that use variants to be interpreted code, because, as with interpreted code, many operations cannot be resolved until run time. This affects in particular the speed of the code.

Now that I've warned you against the use of the Variant type, it is time to look at what it can do. Basically, once you've declared a variant variable such as the following:

```
var
  V: Variant;
```

you can assign to it values of several different types:

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

Once you have the variant value, you can copy it to any compatible-or incompatible-data type. If you assign a value to an incompatible data type, Delphi performs a conversion, if it can. Otherwise it issues a run-time error. In fact, a variant stores type information along with the data, allowing a number of run-time operations; these operations can be handy but are both slow and unsafe.

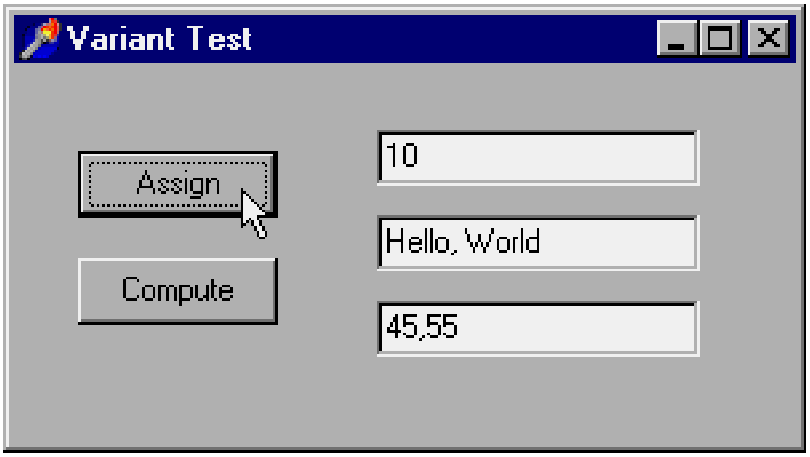
Consider the following example (called VariTest), which is an extension of the code above. I placed three edit boxes on a new form, added a couple of buttons, and then wrote the following code for the OnClick event of the first button:

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  V: Variant;
begin
  V := 10;
  Edit1.Text := V;
  V := 'Hello, World';
  Edit2.Text := V;
  V := 45.55;
  Edit3.Text := V;
end;
```

Funny, isn't it? Besides assigning a variant holding a string to the Text property of an edit component, you can assign to the Text a variant holding an integer or a floating-point number. As you can see in Figure 10.1, everything works.

Figure 10.1: The output of the VariTest example after the Assign button has been pressed.



Even worse, you can use the variants to compute values, as you can see in the code related to the second button:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  V: Variant;
  N: Integer;
begin
  V := Edit1.Text;
  N := Integer(V) * 2;
  V := N;
  Edit1.Text := V;
end;
```

Writing this kind of code is risky, to say the least. If the first edit box contains a number, everything works. If not, an exception is raised. Again, you can write similar code, but without a compelling reason to do so, you shouldn't use the Variant type; stick with the traditional Pascal data types and type-checking approach. In Delphi and in the VCL (Visual Component Library), variants are basically used for OLE support and for accessing database fields.

Variants in Depth

Delphi includes a variant record type, TVarData, which has the same memory layout as the Variant type. You can use this to access the actual type of a variant. The TVarData structure includes the type of the Variant, indicated as VType, some reserved fields, and the actual value.

The possible values of the VType field correspond to the data types you can use in OLE automation, which are often called OLE types or variant types. Here is a complete alphabetical list of the available variant types:

- varArray
- varBoolean
- varByRef
- varCurrency
- varDate
- varDispatch
- varDouble
- varEmpty
- varError
- varInteger
- varNull
- varOleStr
- varSingle
- varSmallint
- varString
- varTypeMask
- varUnknown
- varVariant

You can find descriptions of these types in the Values in variants topic in the Delphi Help system.

There are also many functions for operating on variants that you can use to make specific type conversions or to ask for information about the type of a variant (see, for example, the VarType function). Most of these type conversion and assignment functions are actually called automatically when you write expressions using variants. Other variant support routines (look for the topic Variant support routines in the Help file) actually operate on variant arrays.

Variants Are Slow!

Code that uses the Variant type is slow, not only when you convert data types, but also when you add two variant values holding an integer each. They are almost as slow as the interpreted code of Visual Basic! To compare the speed of an algorithm based on variants with that of the same code based on integers, you can look at the VSpeed example.

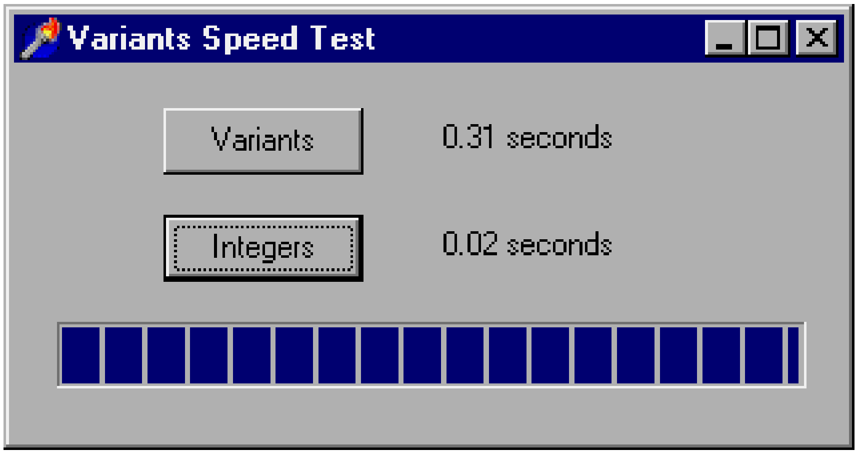
This program runs a loop, timing its speed and showing the status in a progress bar. Here is the first of the two very similar loops, based on integers and variants:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  time1, time2: TDateTime;
  n1, n2: Variant;
begin
  time1 := Now;
  n1 := 0;
  n2 := 0;
  ProgressBar1.Position := 0;
  while n1 < 5000000 do
  begin
    n2 := n2 + n1;
    Inc (n1);
    if (n1 mod 50000) = 0 then
    begin
      ProgressBar1.Position := n1 div 50000;
      Application.ProcessMessages;
    end;
  end;
  // we must use the result
  Total := n2;
  time2 := Now;
  Label1.Caption := FormatDateTime (
    'n:ss', Time2-Time1) + ' seconds';
end;
```

The timing code is worth looking at, because it's something you can easily adapt to any kind of performance test. As you can see, the program uses the Now function to get the current time and the FormatDateTime function to output the time difference, asking only for the minutes ("n") and the seconds ("ss") in the format string. As an alternative, you can use the Windows API's GetTickCount function, which returns a very precise indication of the milliseconds elapsed since the operating system was started.

In this example the speed difference is actually so great that you'll notice it even without a precise timing. Anyway, you can see the results for my own computer in Figure 10.2. The actual values depend on the computer you use to run this program, but the proportion won't change much.

Figure 10.2: The different speeds of the same algorithm, based on integers and variants (the actual timing varies depending on the computer), as shown by the VSpeed example.



Conclusion

Variants are so different from traditional Pascal data types that I've decided to cover them in this short separate chapter. Although their role is in OLE programming, they can be handy to write quick and dirty programs without having even to think about data types. As we have seen, this affects performance by far.

Now that we have covered most of the language features, let me discuss the overall structure of a program and the modularization offered by units.

Next Chapter: Program and Units

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Chapter 11 Program and Units

Delphi applications make intensive use of units, or program modules. Units, in fact, were the basis of the modularity in the language before classes were introduced. In a Delphi application, every form has a corresponding unit behind it. When you add a new form to a project (with the corresponding toolbar button or the File > New Form menu command), Delphi actually adds a new unit, which defines the class for the new form.

Units

Although every form is defined in a unit, the reverse is not true. Units do not need to define forms; they can simply define and make available a collection of routines. By selecting the File > New menu command and then the Unit icon in the New page of the Object Repository, you add a new blank unit to the current project. This blank unit contains the following code, delimiting the sections a unit is divided into:

```
unit Unit1;

interface

implementation

end.
```

The concept of a unit is simple. A unit has a unique name corresponding to its filename, an interface section declaring what is visible to other units, and an implementation section with the real code and other hidden declarations. Finally, the unit can have an optional initialization section with some startup code, to be executed when the program is loaded into memory; it can also have an optional finalization section, to be executed on program termination.

The general structure of a unit, with all its possible sections, is the following:

```
unit unitName;

interface

// other units we need to refer to
uses
  A, B, C;

// exported type definition
type
```

```
  newType = TypeDefinition;

// exported constants
const
  Zero = 0;

// global variables
var
  Total: Integer;

// list of exported functions and procedures
procedure MyProc;

implementation

uses
  D, E;

// hidden global variable
var
  PartialTotal: Integer;

// all the exported functions must be coded
procedure MyProc;
begin
  // ... code of procedure MyProc
end;

initialization
  // optional initialization part

finalization
  // optional clean-up code

end.
```

The uses clause at the beginning of the interface section indicates which other units we need to access in the interface portion of the unit. This includes the units that define the data types we refer to in the definition of other data types, such as the components used within a form we are defining.

The second uses clause, at the beginning of the implementation section, indicates more units we need to access only in the implementation code. When you need to refer to other units from the code of the routines and methods, you should add elements in this second uses clause instead of the first one. All the units you refer to must be present in the project directory or in a directory of the search path (you can set the search path for a project in the Directories/Conditionals page of the project's Options dialog box).

C++ programmers should be aware that the uses statement does not correspond to an include directive. The effect of a uses statement is to import just the precompiled interface portion of the units listed. The implementation portion of the unit is considered only when that unit is compiled. The units you refer to can be both in source code format (PAS) or compiled format (DCU), but the compilation must have taken place with the same version of the Delphi.

The interface of a unit can declare a number of different elements, including procedures, functions, global variables, and data types. In Delphi applications, the data types are probably used the most often. Delphi automatically places a new class data type in a unit each time you create a form. However, containing form definitions is certainly not the only use for units in Delphi. You can continue to have traditional units, with functions and procedures, and you can have units with classes that do not refer to forms or other visual elements.

Units and Scope

In Pascal, units are the key to encapsulation and visibility, and they are probably even more important than the private and public keywords of a class. (In fact, as we'll see in the next chapter, the effect of the private keyword is related to the scope of the unit containing the class.) The scope of an identifier (such as a variable, procedure, function, or a data type) is the portion of the code in which the identifier is accessible. The basic rule is that an identifier is meaningful only within its scope—that is, only within the block in which it is declared. You cannot use an identifier outside its scope. Here are some examples.

- Local variables: If you declare a variable within the block defining a routine or a method, you cannot use this variable outside that procedure. The scope of the identifier spans the whole procedure, including nested routines (unless an identifier with the same name in the nested routine hides the outer definition). The memory for this variable is allocated on the stack when the program executes the routine defining it. As soon as the routine terminates, the memory on the stack is automatically released.
- Global hidden variables: If you declare an identifier in the implementation portion of a unit, you cannot use it outside the unit, but you can use it in any block and procedure defined within the unit. The memory for this variable is allocated as soon as the program starts and exists until it terminates. You can use the initialization section of the unit to provide a specific initial value.
- Global variables: If you declare an identifier in the interface portion of the unit, its scope extends to any other unit that uses the one declaring it. This variable uses memory and has the same lifetime as the previous group; the only difference is in its visibility.

Any declarations in the interface portion of a unit are accessible from any part of the program that includes the unit in its uses clause. Variables of form classes are declared in the same way, so that you can refer to a form (and its public fields, methods, properties, and components) from the code of any other form. Of course, it's poor programming practice to declare everything as global. Besides the obvious memory consumption problems, using global variables makes a program less easy to maintain and update. In short, you should use the smallest possible number of global variables.

Units as Namespaces

The uses statement is the standard technique to access the scope of another unit. At that point you can access the definitions of the unit. But it might happen that two units you refer to declare the same identifier; that is, you might have two classes or two routines with the same name.

In this case you can simply use the unit name to prefix the name of the type or routine defined in the unit. For example, you can refer to the ComputeTotal procedure defined in the given Totals unit as Totals.ComputeTotal. This should not be required very often, as you are strongly advised against using the same name for two different things in a program.

However, if you look into the VCL library and the Windows files, you'll find that some Delphi functions have the same name as (but generally different parameters than) some Windows API functions available in Delphi itself. An example is the simple Beep procedure.

If you create a new Delphi program, add a button, and write the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Beep;
end;
```

then as soon as you press the button you'll hear a short sound. Now, move to the uses statement of the unit and change the code from this:

```
uses
  Windows, Messages, SysUtils, Classes, ...
```

to this very similar version (simply moving the SysUtils unit before the Windows unit):

```
uses
  SysUtils, Windows, Messages, Classes, ...
```

If you now try to recompile this code, you'll get a compiler error: "Not enough actual parameters." The problem is that the Windows unit defines another Beep function with two parameters. Stated more generally, what happens in the definitions of the first units you include in the uses statement might be hidden by corresponding definitions of later units. The safe solution is actually quite simple:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SysUtils.Beep;
end;
```

This code will compile regardless of the order of the units in the uses statements. There are few other name clashes in Delphi, simply because Delphi code is generally hosted by methods of classes. Having two methods with the same name in two different classes doesn't create any problem. The problems arise only with global routines.

Units and Programs

A Delphi application consists of two kinds of source code files: one or more units and one program file. The units can be considered secondary files, which are referred to by the main part of the application, the program. In theory, this is true. In practice, the program file is usually an automatically generated file with a limited role. It simply needs to start up the program, running the main form. The code of the program file, or Delphi project file (DPR), can be edited either manually or by using the Project Manager and some of the Project Options related to the application object and the forms.

The structure of the program file is usually much simpler than the structure of the units. Here is the source code of a sample program file:

```
program Project1;

uses
  Forms,
  Unit1 in ‘Unit1.PAS’ {Form1DateForm};

begin
  Application.Initialize;
  Application.CreateForm (TForm1, Form1);
  Application.Run;
end.
```

As you can see, there is simply a uses section and the main code of the application, enclosed by the begin and end keywords. The program's uses statement is particularly important, because it is used to manage the compilation and linking of the application.

Conclusion

At least for the moment, this chapter on the structure of a Pascal application written in Delphi or with one of the latest versions of Turbo Pascal, is the last of the book. Feel free to email me your comment and requests.

If after this introduction on the Pascal language you want to delve into the object-oriented elements of Object Pascal in Delphi, you can refer to my published book **Mastering Delphi 5** (Sybex, 1999). For more information on this and more advanced books of mine (and of other authors as well) you can refer to my web site, **www.marcocantu.com**. The same site hosts updated versions of this book, and its examples.

Back to the Cover Page

- www.marcocantu.com
- Marco's Delphi Books
- Essential Pascal - Web Site
- Essential Pascal - Local Index

Marco Cantù's Essential Pascal

Appendix A Glossary

This is a short glossary of technical terms used throughout the book. They might also be defined elsewhere in the text, but I've decided to collect them here anyway, to make it easier to find them.

Heap (Memory)

The term Heap indicates a portion of the memory available to a program, also called dynamic memory area. The heap is the area in which the allocation and deallocation of memory happens in random order. This means that if you allocate three blocks of memory in sequence, they can be destroyed later on in any order. The heap manager takes care of all the details for you, so you simply ask for new memory with GetMem or by calling a constructor to create an object, and Delphi will return you a new memory block (optionally reusing memory blocks already discarded).

The heap is one of the three memory areas available to an application. The other two are the global area (this is where global variables live) and the stack. Contrary to the heap, global variables are allocated when the program starts and remain there until it terminates. For the stack see the specific entry in this glossary.

Delphi uses the heap for allocating the memory of each and every object, the text of the strings, for dynamic arrays, and for specific requests of dynamic memory (GetMem).

Windows allows an application to have up to 2 GigaBytes of address space, most of which can be used by the heap.

Stack (Memory)

The term Stack indicates a portion of the memory available to a program, which is dynamic but is allocated and deallocated following specific order. The stack allocation is LIFO, Last In First Out. This means that the last memory object you've allocated will be the first to be deleted. Stack memory is typically used by routines (procedure, function, and method calls). When you call a routine, its parameters and return type are placed on the stack (unless you optimize the call, as Delphi does by default). Also the variables you declare within a routine (using a var block before the begin statement) are stored on the stack, so that when the routine terminates they'll be automatically removed (before getting back to the calling routine, in LIFO order).

The stack is one of the three memory areas available to an application. The other two are called global memory and heap. See the heap entry in this glossary..

Delphi uses the stack for routine parameters and return values (unless you use the default register calling convention), for local routine variables, for Windows API function calls, and so on.

Windows applications can reserve a large amount of memory for the stack. In Delphi you set this in the linker page of the project options, however, the default generally does it. If you receive a stack full error message this is probably because you have a function recursively calling itself forever, not because the stack space is too limited.

New requested terms

- Dynamic
- Static
- Virtual
- memory leak
- painting
- literal
- array
- API
- class reference
- class method
- parent
- owner
- self

MARCO CANTÙ'S ESSENTIAL DELPHI

A Friendly Introductory Guide to Borland Delphi

http://www.marcocantu.com/edelphi

Revision 1.03 - April 13, 2002

INTRODUCTION

After the successful publishing of the e-book Essential Pascal (available on my web site at the address `http://www.marcocantu.com/epascal`), I decided to follow up with an introduction to Delphi. Again most of the material you'll find here was in the first editions of my "printed" book Mastering Delphi, the best selling Delphi book I have written. Due to space constraints and because many Delphi programmers look for more advanced information, in the latest edition this material was completely omitted. To overcome the absence of this information, I have started putting together this second on-line book, titled ***Essential Delphi***.

Copyright

The text and the source code of this book are copyrighted by Marco Cantù. Of course, you can use the programs and adapt them to your own needs with no limitation, only you are not allowed to use them in books, training material, and other copyrighted formats without my permission (or in case you are using limited portions, referring to the original). Feel free to link your site with this one, but please do not duplicate the material (on your web site, on a CD) as it is subject to frequent changes and updates. Passing a copy to a friend, occasionally, is certainly something you can do if you do not modify it in any way.

You can print out this book both for personal use and for non-profit training (user-groups, schools, and universities are free to distribute a printed versions as long as they don't charge more than the printing costs and make it clear that this material is freely available, referring readers to the Essential Delphi web site (`http://www.marcocantu.com/edelphi`) for updates.

Book Structure

The book structure is still under development, as the book evolves. This is the current structure:

Chapter 1: A Form is a Window:
Chapter 2: Highlights of the Delphi Environment:

Chapter 3: The Object Repository and the Delphi Wizards:

Chapter 4: A Tour of the Basic Components

Chapter 5: Creating and Handling Menus [some figures still missing]

Chapter 6: Multimedia Fun [all figures missing]

Planned chapters:

Chapter 7: Exploring Forms

Chapter 8: Delphi Database 101

Chapter 9: Reporting Basics

Source Code

The source code of all the examples mentioned in the book is available on the book web site. The code has the same Copyright as the book: Feel free to use it at will but don't publish it on other documents or site. Links back to this site are welcome.

Feedback

Please let me know of any errors you find (indicating revision number and page number), but also of topics not clear enough for a beginner. I'll be able to devote time to the project depending also on the feedback I receive. Let me know also which other topics (not covered in Mastering Delphi) you'd like to see here.

For reporting errors please use the books section of my newsgroup, as described on www.marcocantu.com or use my mailbox (which gets far too jammed) at marco@marcocantu.com.

Acknowledgments

I have first started thinking about on-line publishing after Bruce Eckel's experience with *Thinking in Java*. I'm a friend of Bruce and think he really did a great job with that book and few others. After the overwhelming response of the "Essential Pascal" book, I started this new one and plan releasing the two as a printed book introducing Delphi (the only problem being to find a publisher).

About the Author

Marco Cantù lives in Piacenza, Italy. After writing C++ and Object Windows Library books and articles, he delved into Delphi programming. He is the author of the *Mastering Delphi* book series, published by Sybex, as well as the advanced *Delphi Developers Handbook*. He writes articles for many magazines, including *The Delphi Magazine*, speaks at Delphi and Borland conferences around the world, and teaches Delphi classes at basic and advanced levels. More recently, he's specializing in XML technologies, still making most of his programming in Delphi. Of course, you can learn more details about Marco and his work by visiting his web site, www.marcocantu.com.

Donations

I'll probably set up an account on one of those donation/contribution systems, to let people who have enjoyed the book and learned from it, particularly if programming is their job (and not a hobby) and they do it for profit, contribute to its development. No extra material is offered to those donating to the book fund, only because I want to let anyone (particularly students and people leaving in poor countries) benefit from the availability of this material. Information will be available on the book web site.

Table of Contents

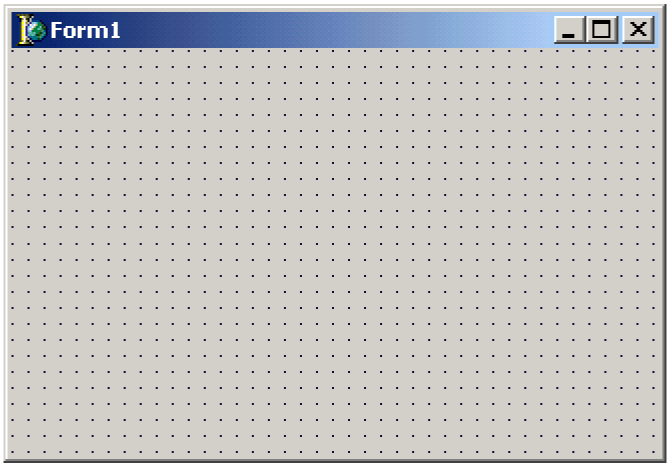
Chapter 1: A Form Is a Window

Windows applications are usually based on windows. So, how are we going to create our first window? We'll do it by using a form. As the first part of the title suggests, a form really is a window in disguise. There is no real difference between the two concepts, at least from a general point of view.

> If you look closely, a form is always a window, but the reverse isn't always true. Some Delphi components are windows, too. A push button is a window. A list box is a window. To avoid confusion, I'll use the term *form* to indicate the main window of an application or a similar window and the term *window* in the broader sense.

Creating Your First Form

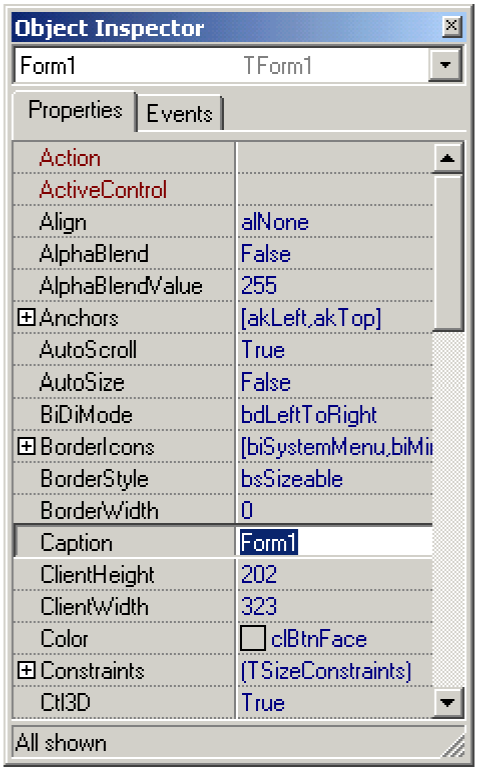
Even though you have probably already created at least some simple applications in Delphi, I'm going to show you the process again, to highlight some interesting points. Creating a form is one of the easiest operations in the system: you only need to open Delphi, and it will automatically create a new, empty form for you, as you can see in the figure below. That's all there is to it.



If you already have another project open, choose File | New | Application to close the old project (you may be prompted to save some of the files) and open a new blank project. Believe it or not, you already have a working application. You can run it, using the Run button on the toolbar or the Run | Run menu command, and it will result in a standard Windows program. Of course, this application won't be very useful, since it has a single empty window with no capabilities, but the default behavior of any Windows window.

Adding a Title

Before we run the application, let's make a quick change. The title of the form is *Form1*. For a user, the title of the main window stands for the name of the application. Let's change *Form1* to something more meaningful. When you first open Delphi, the Object Inspector window should appear on the left side of the form (if it doesn't, open it by choosing View | Object Inspector or pressing the F11 key):



Action	
ActiveControl	
Align	alNone
AlphaBlend	False
AlphaBlendValue	255
⊞Anchors	[akLeft,akTop]
AutoScroll	True
AutoSize	False
BiDiMode	bdLeftToRight
⊞BorderIcons	[biSystemMenu,biMi
BorderStyle	bsSizeable
BorderWidth	0
Caption	Form1
ClientHeight	202
ClientWidth	323
Color	□ clBtnFace
⊞Constraints	(TSizeConstraints)
Ctl3D	True

The Object Inspector shows the properties of the selected component. The window contains a tab control with two pages. The first page is labeled Properties. The other page is labeled Events and shows a list of events that can take place in the form or in the selected component.

The properties are listed in alphabetical order, so it's quite easy to find the ones you want to change (it is also possible to group them by category, as we'll see in the next chapter, but this feature is seldom used by Delphi developers). We can change the title of the form simply by changing the `Caption` property, which is selected by default. While you type a new caption, you can see the title of the form change. If you type *Hello*, the title of the form changes immediately. As an alternative, you can modify the internal name of the form by changing its `Name` property. If you have not entered a new caption, the new value of the `Name` property will be used for the `Caption` property, too.

Only a few of the properties of a component change while you type the new value. Most are applied when you finish the editing operation and press the Enter key (or move the input focus to a new property).

Although we haven't done much work, we have built a full-blown application, with a system menu and the default Minimize, Maximize, and Close buttons. You can resize the form by dragging its borders, move it by dragging its caption, maximize it to full-screen size, or minimize it. It works, but again, it's not very useful. If you look at the icon in the Taskbar, you'll see that something isn't right. Instead of showing the caption of the form as the icon caption, it shows the name of the project, something like *Project1*. We can fix this by giving a name to the project, which we'll do by saving it to disk with a new name.

Saving the Form

Select the Save Project or Save Project As command from the File menu, and Delphi will ask you to give a name to the source code file associated with the form, and then to name the project file. Since the name of the project should match the caption of the form (*Hello*), I've named the form source file `HELLOF.PAS`, which stands for *Hello Form*. I've given the project file the name `HELLO.DPR`.

Unfortunately, we cannot use the same name for the project and the unit that defines the form; for each application, these items must have unique names. You can add the letter *F*, add *Form*, call every form unit *MainForm*, or choose any other naming convention you like. I tend to use a name similar to the project name, as simply calling it `Mainform` means you'll end up with a number of forms (in different projects) that all have the same name.

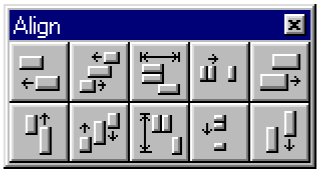
The name you give to the project file is used by default at run-time as the title of the application, displayed by Windows in the taskbar while the program is running. For this reason, if the name of the project matches the caption of the main form, it will also correspond to the name on the taskbar. You can also change the title of the application by using the Application page of the Project Options dialog box (choose Project | Options), or by writing a line of code to change the `Title` property of the Application global object.

Using Components

Now it's time to start placing something useful in our Hello form. Forms can be thought of as component containers. Each form can host a number of components or controls. You can choose a component from the Components Palette above the form, in the Delphi window. There are four simple ways to place a component on a form. If you choose the Button component from the Standard page of the Components Palette, for example, you can do any of the following:

- Click on the component, move the mouse cursor to the form, press the left mouse button to set the upper-left corner of the button, and drag the mouse to set the button's size.
- Select the component as above, and then simply click on the form to place a button of the default height and width.
- Double-click on the icon in the Components Palette, and a component of that type will be added in the center of the form.
- Shift-click on the component icon, and place several components of the same kind in the form using one of the above procedures.

Our form will have only one button, so we'll center it in the form. You can do this by hand, with a little help from Delphi. When you choose View | Alignment Palette, a toolbox with alignment icons appears:



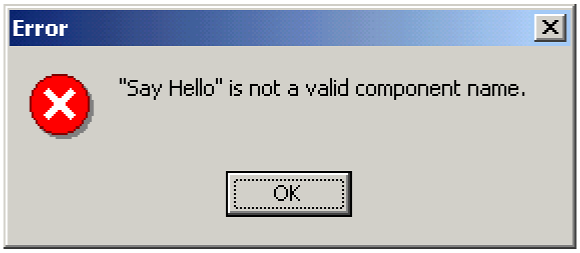
This toolbox makes a number of operations easy. It includes buttons to align controls or to center them in the form. Using the two buttons in the third column, you can place a component in the center of the form. Although we've placed the button in the center, as soon as you run the program, you can resize the form so that the button won't be in the center anymore. So the button is only in the center of the form at startup. Later on, we'll see how to make the button remain in the center after the form is resized, by adding some code. For now, our first priority is to change the button's label.

Changing Properties

Like the form, the button has a `Caption` property that we can use to change its label (the text displayed inside it). As a better alternative, we can change the name of the button. The name is a kind of internal property, used only in the code of the program. However, as I mentioned earlier, if you change the name of a button before changing its caption, the `Caption` property will have the same text as the `Name` property. Changing the `Name` property is usually a good choice, and you should generally do this early in the development cycle, before you write much code.

> It is quite common to define a naming convention for each type of component (usually the full name or a shorter version, such as "btn" for Button). If you use a different prefix for each type of component (as in "*ButtonHello*" or "*BtnHello*"), the combo box above the Object Inspector will list the components of the same kind in a group, because they are alphabetically sorted. If you instead use a suffix, naming the components "*HelloButton*" or "*HelloBtn*," components of the same kind will be in different positions on the list. In this second case, however, finding a particular component using the keyboard might be faster. In fact, when the Object Inspector is selected you can type a letter to jump to the first component whose name starts with that letter.

Besides setting a proper name for a component, you often need to change its `Caption` property. There are at least two reasons to have a caption different from the name. The first is that the name often follows a naming convention (as described in the note above) that you won't want to use in a caption. The second reason is that captions should be descriptive, and therefore they often use two or more words, as in my *Say hello* button. If you try to use this text as the `Name` property, however, Delphi will show an error message:



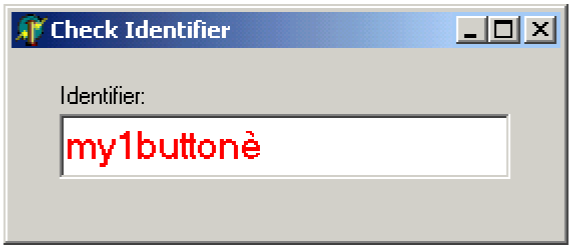
Error

"Say Hello" is not a valid component name.

The name is an internal property, and it is used as the name of a variable referring to the component. Therefore, for the `Name` property, you must follow the rules for naming an identifier in the Pascal language:

- An identifier is a sequence of letters, digits, or underscore characters of any length (although only the first 63 characters are significant).
- The first character of an identifier cannot be a number; it must be a letter or the underscore character.
- No spaces are allowed in an identifier.
- Identifiers are not case-sensitive, but usually each word in an identifier begins with a capital letter, as in BtnHello. But btnhello, btnHello, and BTNHello refer to this same identifier.

You can use the `IsValidIdent` system function to check whether a given string is a valid identifier. The CheckId example calls this function while you type an identifier in its edit box, and changes the text color to indicate whether the string is valid (green) or not (red). The code of the example is quite simple, and you can look at it yourself on the disk. Try running this program to check any doubts about allowed component names.



Check Identifier

Identifier:

my1buttonè

Here is a summary of the changes we have made to the properties of the button and form. At times, I'll show you the structure of the form of the examples as it appears once it has been converted in a readable format (I'll describe how to convert a form into text later in this chapter). I won't show you the entire textual description of a form (which is often quite long), but rather only its key elements. I won't include the lines describing the position of the components, their sizes, or some less important default values. Here is the code:

```
object Form1: TForm1
  Caption = 'Hello'
  OnClick = FormClick
  object BtnHello: TButton
    Caption = 'Say hello'
    OnClick = BtnHelloClick
  end
```

```
end
```

This description shows some attributes of the components and the events they respond to. We will see the code for these events in the following sections. If you run this program now, you will see that the button works properly. In fact, if you click on it, it will be pushed, and when you release the mouse button, the on-screen button will be released. The only problem is that when you press the button, you might expect something to happen; but nothing does, because we haven't assigned any action to the mouse-click yet.

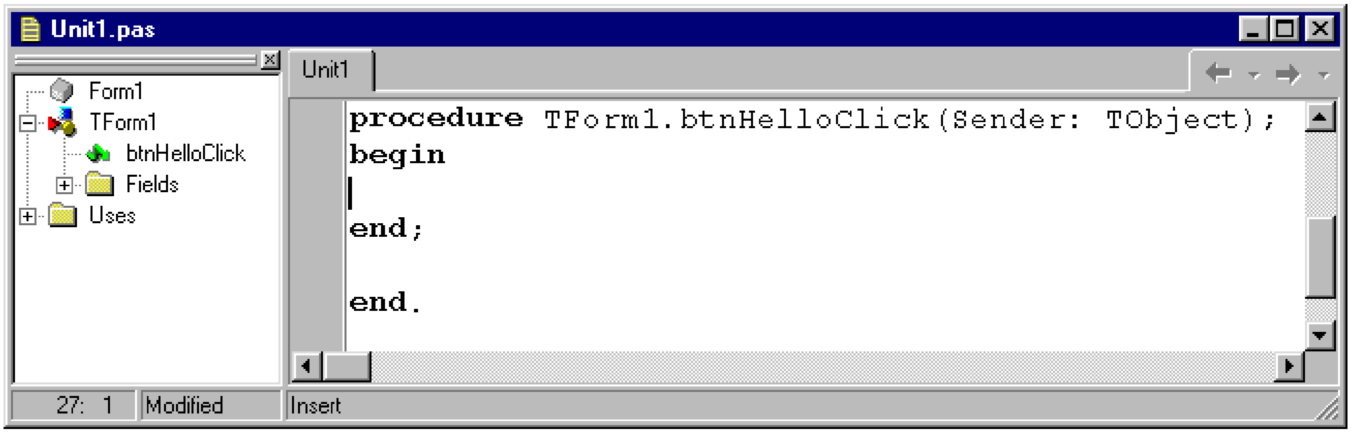
Responding to Events

When you press the mouse button on a form or a component, Windows informs your application of the event by sending it a message. Delphi responds by receiving an event notification and calling the appropriate event-handler method. As a programmer, you can provide several of these methods, both for the form itself and for the components you have placed in it. Delphi defines a number of events for each kind of component. The list of events for a form is different from the list for a button, as you can easily see by clicking on these two components while the Events page is selected in the Object Inspector. Some events are common to both components.

There are several techniques you can use to define a handler for the `OnClick` event of the button:

- Select the button, either in the form or by using the Object Inspector's combo box (called the Object Selector), select the Events page, and double-click in the white area on the right side of the OnClick event. A new method name will appear, *BtnHelloClick*.
- Select the button, select the Events page, and enter the name of a new method in the white area on the right side of the `OnClick` event. Then press the Enter key to accept it.
- Double-click on the button, and Delphi will perform the default action for this component, which is to add a handler for the `OnClick` event. Other components have completely different default actions.

With any of these approaches, Delphi creates a procedure named `BtnHelloClick` (or the name you've provided) in the code of the form and opens the source code file in that position:



```
procedure TForm1.btnHelloClick(Sender: TObject);
begin

end;

end.
```

The default action for a button is to add a procedure to respond to the click event. Even if you are not sure of the effect of the default action of a component, you can still double-click on it. If you end up adding a new procedure you don't need, just leave it empty. Empty method bodies generated by Delphi will be removed as soon as you save the file. In other words, if you don't put any code in them, they simply go away.

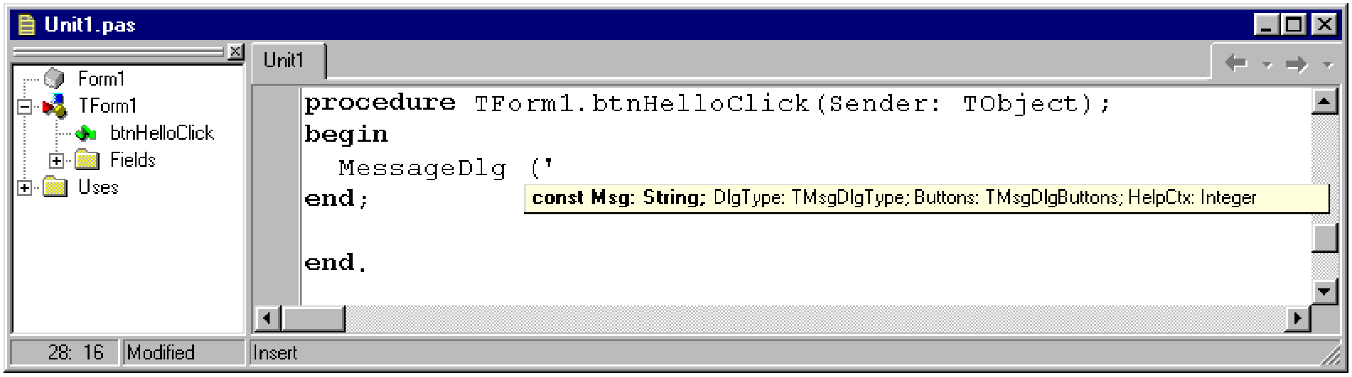
> When you want to remove an event-response method you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event-handler that is still empty, consider adding a comment to it, so that it will not be removed.

Now we can start typing some instructions between the `begin` and `end` keywords that delimit the code of the procedure. Writing code is usually so simple that you don't need to be an expert in the language to start working with Delphi. (If you need to brush up your knowledge of Pascal you can refer to my online Essential Pascal book, while if you need derailede coverage of Object Pascal you can refer to my Mastering Delphi series.)

You should type only the line in the middle, but I've included the whole source code of the procedure to let you know where you need to add the new code in the editor:

```
procedure TForm1.BtnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
end;
```

The code is simple. There is only a call to a function, `MessageDlg`, to display a small message dialog box. The function has four parameters. Notice that as you type the open parenthesis, the Delphi editor will show you the list of parameters in a hint window, making it simpler to remember them.



```
procedure TForm1.btnHelloClick(Sender: TObject);
begin
  MessageDlg ('
end;

end.
```

const Msg: String; DlgType: TMsgDlgType; Buttons: TMsgDlgButtons; HelpCtx: Integer

If you need more information about the parameters of this function and their meanings, you can click on its name in the edit window and press F1. This brings up the Help information. Since this is the first code we are writing, here is a summary of that description (the rest of this book, however, generally does *not* duplicate the reference information available in Delphi's Help system, concentrating instead on examples that demonstrate the features of the language and environment):

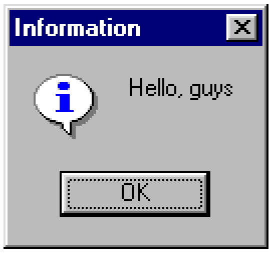
- The first parameter of the MessageDlg function is the string you want to display: the message.

- The second parameter is the type of message box. You can choose mtWarning, mtError, mtInformation, or mtConfirmation. For each type of message, the corresponding caption is used and a proper icon is displayed at the side of the text.
- The third parameter is a set of values indicating the buttons you want to use. You can choose mbYes, mbNo, mbOK, mbCancel, or mbHelp. Since this is a set of values, you can have more than one of these values. Always use the proper set notation with square brackets ([and]) to denote the set, even if you have only one value, as in the line of the code above. (Essential Pascal discusses Pascal sets.)
- The fourth parameter is the help context, a number indicating which page of the Help system should be invoked if the user presses F1. Simply write `0` if the application has no help file, as in this case.

The function also has a return value, which I've just ignored, using it as if it were a procedure. In any case, it's important to know that the function returns an identifier of the button that the user clicked to close the message box. This is useful only if the message box has more than one button.

> Programmers unfamiliar with the Pascal language, particularly those who use C/C++, might be confused by the distinction between a function and a procedure. In Pascal, there are two different keywords to define procedures and functions. The only difference between the two is that functions have a return value.

After you have written this line of code, you should be able to run the program. When you click on the button, you'll see the message box shown below.



Hello, guys

Every time the user clicks on the push button in the form, a message is displayed. What if the mouse is pressed outside that area? Nothing happens. Of course, we can add some new code to handle this event. We only need to add an `OnClick` event to the form itself. To do this, move to the Events page of the Object Inspector and select the form. Then double-click at the right side of the `OnClick` event, and you'll end up in the proper position in the edit window. Now add a new call to the `MessageDlg` function, as in the following code:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;
```

With this new version of the program, if the user clicks on the button, the hello message is displayed, but if the user misses the button, a warning message appears. Notice that I've written the code on two lines, instead of one. The Pascal compiler completely ignores new lines, white spaces, tab spaces, and similar formatting characters. Program statements are separated by semicolons (`;`), not by new lines.

> There is one case in which Delphi doesn't completely ignore line breaks: Strings cannot extend across multiple lines. In some cases, you can split a very long string into two different strings, written on two lines, and merge them by writing one after the other.

Compiling and Running a Program

Before we make any further changes to our Hello program, let's stop for a moment to consider what happens when you run the application. When you click on the toolbar Run button or select Run | Run, Delphi does the following:

1: Compiles the Pascal source code file describing the form.

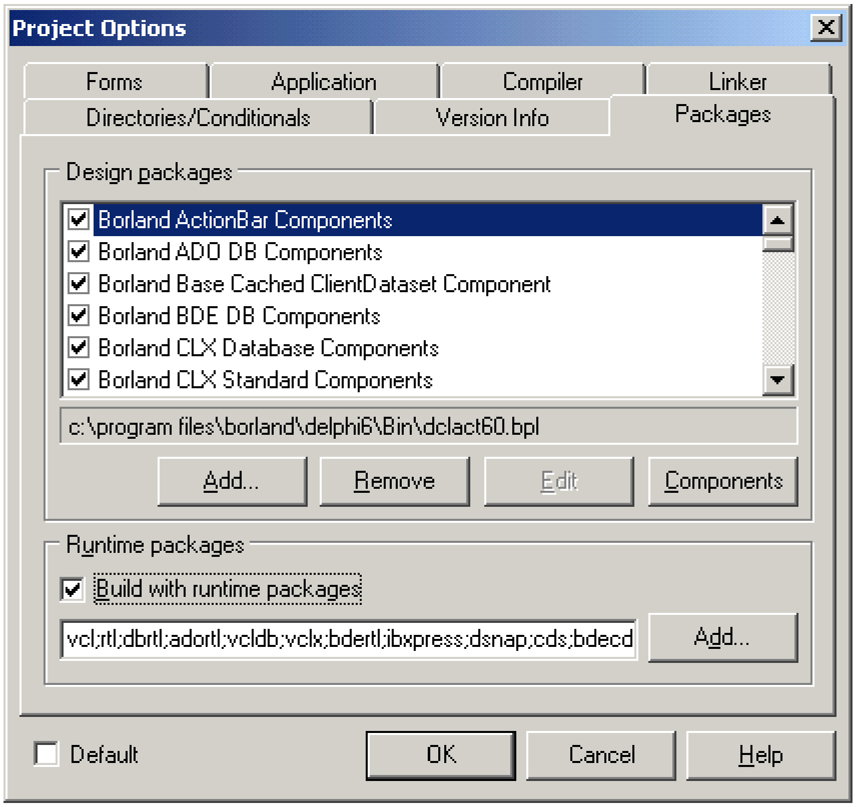
2: Compiles the project file.

3: Builds the executable (`EXE`) file, linking the proper libraries.

4: Runs the executable file, usually in debug mode.

> In early versions of Delphi, the executable file you obtained was invariably a stand-alone program. Starting with version 3, Delphi allows you to link all the required libraries into the executable file, but you can also specify the use of separate run-time packages, making the executable file much smaller.

The key point is that when you ask Delphi to run your application, it compiles it into an executable file. You can easily run this file from the Windows Explorer or using the Run command on the Start button. Compiling this program as usual, linking all the required library code, produces an executable of about a couple of hundred Kb. By using run-time packages, this can shrink the executable to about 20 Kb. Simply select the Project | Options menu command, move to the Packages page, and select the check box *Build with runtime packages*:



☑ Borland ActionBar Components
☑ Borland ADO DB Components
☑ Borland Base Cached ClientDataset Component
☑ Borland BDE DB Components
☑ Borland CLX Database Components
☑ Borland CLX Standard Components

c:\program files\borland\delphi6\Bin\dclact60.bpl

☑ Build with runtime packages

vcl;rtl;dbrtl;adortl;vcldb;vclx;bdertl;ibxpress;dsnap;cds;bdecd

□ Default

Packages are dynamic link libraries containing Delphi components (the Visual Components Library). By using packages you can make an executable file much smaller. However, the program won't run unless the proper dynamic link libraries (such as `vcl60.bpl`) are available on the computer where you want to run the program. The `BPL` extensions stands for Borland Package Libraries; it is the extension used by Delphi (and C++Builder) packages, which are technically DLL files. Using this extension makes it easier to recognize them (and find them on a hard disk).

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the program built with run-time packages is much bigger than the space required by the bigger stand-alone executable file. For this reason the use of packages is not always recommended. The great advantage of Delphi over competing development tools is that you can easily choose whether to use the stand-alone executable or the small executable with run-time packages.

> In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable with that of a C or C++ program. Delphi compiled code runs much faster (at least 10 times faster) than the equivalent code in interpreted or *semi-compiled* tools.

Some users cannot believe that Delphi generates real executable code, because when you run a small program, its main window appears almost immediately, as happens in some interpreted environments. To see for yourself, try this: Open the Environment Options dialog box (using Tools | Options), move to the Preferences page, and turn on the Show Compile Progress option. Now select Project | Build All. You'll see a dialog box with the compilation status. You'll find that this takes just a few seconds, or even less on a fast machine.

In the tradition of Borland's Turbo Pascal compilers, the Object Pascal compiler embedded in Delphi works very quickly. For a number of technical reasons, it is much faster than any C++ compiler. If you try using the new Borland C++ Builder development environment (which is very similar to Delphi) the compilation requires more time, particularly the first time you build an application. One reason for the higher speed of the Delphi compiler is that the language definition is simpler. Another is that the Pascal compilers and linkers have less work to do to include libraries or other compiled source files in a program, because of the structure of units.

Changing Properties at Run-Time

Let's return to the Hello application. We now want to try to change some properties at run-time. For example, we might change the text of `HelloButton` from *Say hello* to *Say hello again* after the first time a user clicks on it. You may also need to widen the button, as the caption becomes longer. This is really simple. You only need to change the code of the `HelloButtonClick` procedure as follows:

```
procedure TForm1.HelloButtonClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  btnHello.Caption := 'Say Hello Again';
end;
```

> The Pascal language uses the `:=` operator to express an assignment and the `=` operator to test for equality. At the beginning, this can be confusing for programmers coming from other languages. For example in C and C++, the assignment operator is `=`, and the equality test is `==`. After a while, you'll get used to it. In the meantime, if you happen to use `=` instead of `:=`, you'll get an error message from the compiler.

A property such as `Caption` can be changed at run-time very easily, by using an assignment statement. Most properties can be changed at run-time, and some can be changed *only* at run-time. You can easily spot this last group: They are not listed in the Object Inspector, but they appear in the Help file for the component. Some of these run-time properties are defined as read-only, which means that you can access their value but cannot change it.

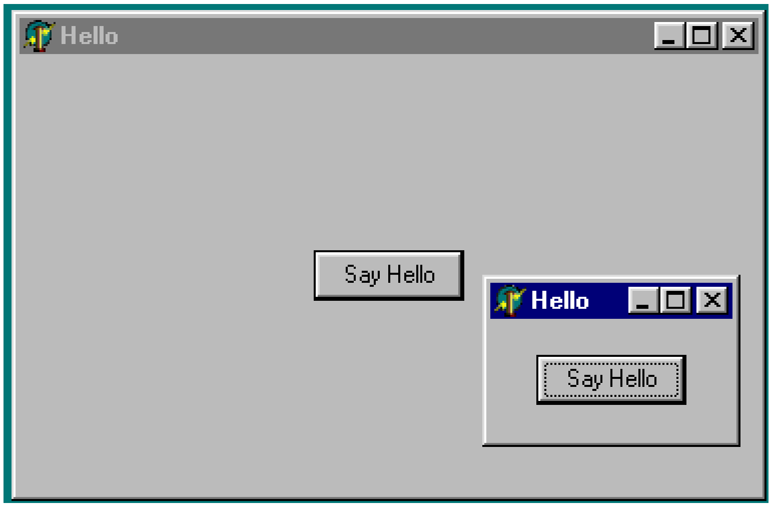
Adding Code to the Program

Our program is almost finished, but we still have a problem to solve, which will require some real coding. The button starts in the center of the form, but will not remain there when you resize the form. This problem can be solved in two radically different ways.

One solution is to change the border of the form to a thin frame, so that the form cannot be resized at run-time. Just move to the `BorderStyle` property of the form, and choose `bsSingle` instead of `bsSizeable` from the combo box. The other approach is to write some code to move the button to the center of the form each time the form is resized, and that's what we'll do next. Although it might seem that most of your work in programming with Delphi is just a matter of selecting options and visual elements, there comes a time when you need to write code. As you become more expert, the percentage of the time spent writing code will generally increase.

When you want to add some code to a program, the first question you need to ask yourself is Where? In an event-driven environment, the code is always executed in response to an event. When a form is resized, an event takes place: `OnResize`. Select the form in the Object Inspector and double-click next to `OnResize` in the Events page. A new procedure is added to the source file of the form. Now you need to type some code in the editor, as follows:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  BtnHello.Top := Form1.ClientHeight div 2 -
    BtnHello.Height div 2;
  BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;
```



To set the `Top` and `Left` properties of the button — that is, the position of its upper-left corner — the program computes the center of the frame, dividing the height and the width of the internal area or client area of the frame by 2, and then subtracts half the height or width of the button. Note also that if you use the `Height` and `Width` properties of the form, instead of the `ClientWidth` and `ClientHeight` properties, you will refer to the center of the whole window, including the caption at the top border. This final version of the example works quite well as you can see below.

This figure includes two versions of the form, with different sizes. By the way, this figure is a real snapshot of the screen. Once you have created a Windows application, you can run several copies of it at the same time by using the Explorer. By contrast, the Delphi environment can run only one copy of a program. When you run a program within Delphi, you start the integrated debugger, and it cannot debug two programs at the same time — not even two copies of the same program — unless you are using Windows NT/2000/XP.

A Two-Way Tool

In the Hello example, we have written three small portions of code, to respond to three different events. Each portion of code was part of a different procedure (actually a method, as you'll learn reading Chapter 5). But where does the code we write end up? The source code of a form is written in a single Pascal language source file, the one we've named HELLOF.PAS. This file evolves and grows not only when you code the response of some events, but also as you add components to the form. The properties of these components are stored together with the properties of the form in a second file, named HELLOF.DFM.

Delphi can be defined as a two-way tool, since everything you do in the visual environment ends up in some code. Nothing is hidden away and inaccessible. You have the complete code, and although some of it might be fairly complex, you can edit everything. Of course, it is easier to use only the visual tools, at least until you are an expert Delphi programmer.

The term *two-way tool* also means that you are free to change the code that has been produced, and then go back to the visual tools. This is true as long as you follow some simple rules.

Looking at the Source Code

Let's take a look at what Delphi has generated from our operations so far. Every action has an effect — in the Pascal code, in the code of the form, or in both. When you start a new, blank project, the empty form has some code associated with it, as in the following listing.

```
unit Unit1;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

The file, named Unit1, uses a number of units and defines a new data type (a class) and a new variable (an object of that class). The class is named TForm1, and it is derived from TForm. The object is Form1, of the new type TForm1.

> *Units* are the modules into which a Pascal program is divided. When you start a new project, Delphi generates a program module and a unit that defines the main form. Each time you add a form to a Delphi program, you add a new unit. Units are then compiled separately and linked into the main program. By default, unit files have a .PAS extension and program files have a .DPR extension.

If you rename the files as suggested in the example, the code changes slightly, since the name of the unit must reflect the name of the file. If you name the file Hellof.pas, the code begins with

```
unit Hellof;
```

As soon as you start adding new components, the form class declaration in the source code changes. For example, when you add a button to the form, the portion of the source code defining the new data type becomes the following:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    ...
```

Now if you change the button's Name property (using the Object Inspector) to BtnHello, the code changes slightly again:

```
type
  TForm1 = class(TForm)
    BtnHello: TButton;
    ...
```

Setting properties other than the name has no effect in the source code. The properties of the form and its components are stored in a separate form description file (with a DFM extension).

Adding new event handlers has the biggest impact on the code. Each time you define a new handler for an event, a line is added to the data type definition of the form, an empty method body is added in the implementation part, and some information is stored in the form description file, too.

```
unit HelloForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    btnHello: TButton;
    procedure btnHelloClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.btnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  btnHello.Caption := 'Say Hello Again';
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;

procedure TForm1.FormResize(Sender: TObject);
begin
  BtnHello.Top := Form1.ClientHeight div 2 -
    BtnHello.Height div 2;
  BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;

end.
```

It is worth noting that there is a single file for the whole code of the form, not just small fragments. Of course, the code is only a partial description of the form. The source code determines how the form and its components react to events. The form description (the `DFM` file) stores the values of the properties of the form and of its components. In general, source code defines the actions of the system, and form files define the initial state of the system.

The Textual Description of the Form

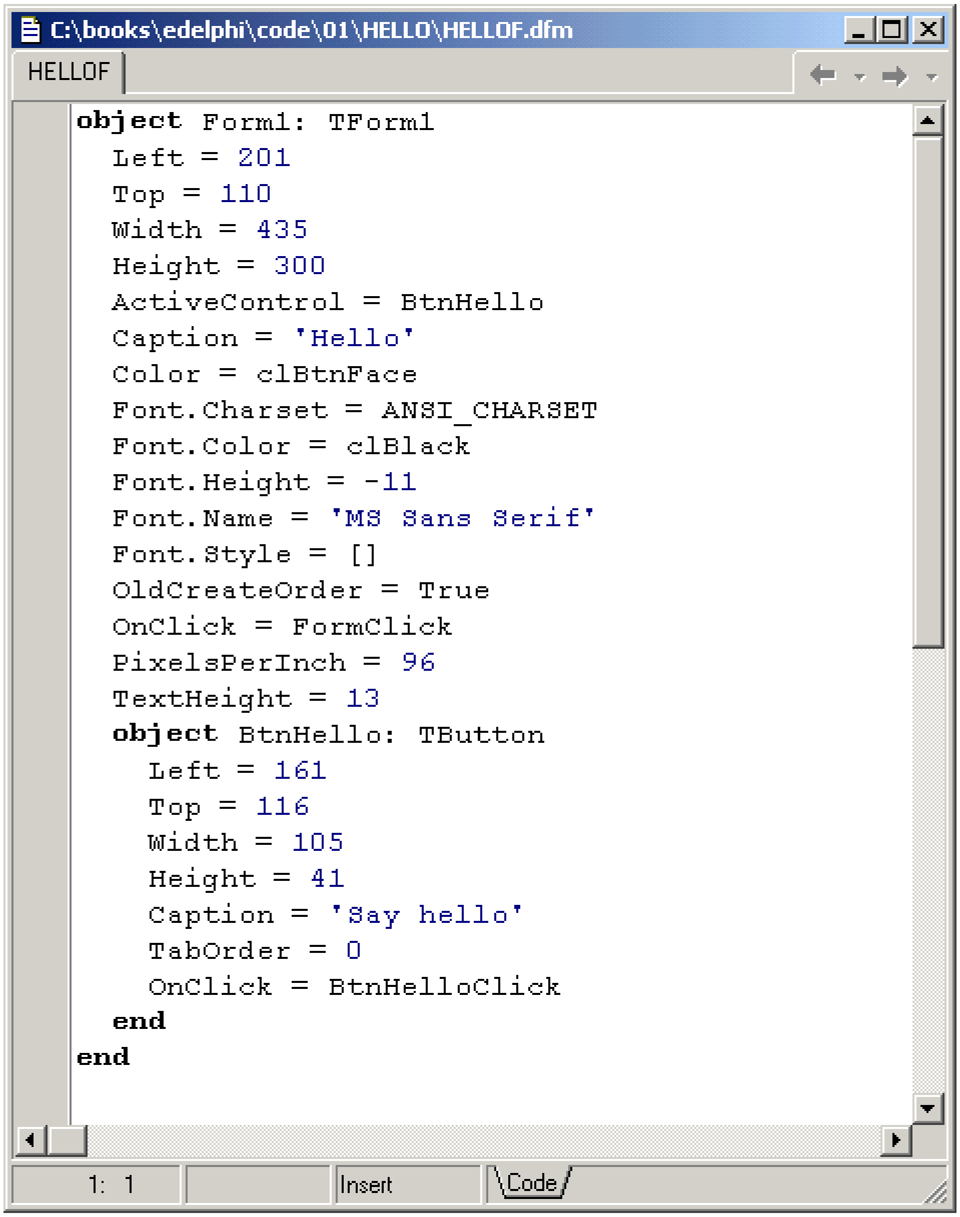
As I've just mentioned, along with the `PAS` file containing the source code, there is another file describing the form, its properties, its components, and the properties of the components. This is the `DFM` file, a binary or text file (this latter option has been introduced with Delphi 5). Whatever the format, if you load this file in the Delphi code editor, it will be converted into a textual description. This might give the false impression that the `DFM` file is indeed a text file, but this can be only if you've selected the corresponding option (available since Delphi 5).

> You can open the textual description of a form simply by selecting the shortcut menu of the form designer (that is, right-clicking on the surface of the form at design-time) and selecting the View as Text command. This closes the form, saving it if necessary, and opens the `DFM` file in the editor. You can later go back to the form using the View as Form command of the local menu of the editor window. The alternative is to open the `DFM` file directly in the Delphi editor.

To understand what is stored in the `DFM` file, you can look at the next listing, which shows the textual description of the form of the first version of the Hello example. This is exactly the code you'll see if you give the View as Text command in the local menu of the form:

```
object Form1: TForm1
  Left = 235
  Top = 108
  Width = 430
  Height = 308
  Caption = 'Hello'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  OnCreate = FormCreate
  OnResize = FormResize
  PixelsPerInch = 96
  TextHeight = 13
  object btnHello: TButton
    Left = 165
    Top = 111
    Width = 75
    Height = 25
    Caption = 'Say Hello'
    TabOrder = 0
    OnClick = btnHelloClick
  end
end
```

You can compare this code with what I used before to indicate the key features and properties of the form and its components. As you can see in this listing, the textual description of a form contains a number of objects (in this case, two) at different levels. The `Form1` object contains the `BtnHello` object, as you can immediately see from the indentation of the text. Each object has a number of properties, and some methods connected to events (in this case, `OnClick`).



```
object Form1: TForm1
  Left = 201
  Top = 110
  Width = 435
  Height = 300
  ActiveControl = BtnHello
  Caption = 'Hello'
  Color = clBtnFace
  Font.Charset = ANSI_CHARSET
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = True
  OnClick = FormClick
  PixelsPerInch = 96
  TextHeight = 13
  object BtnHello: TButton
    Left = 161
    Top = 116
    Width = 105
    Height = 41
    Caption = 'Say hello'
    TabOrder = 0
    OnClick = BtnHelloClick
  end
end
```

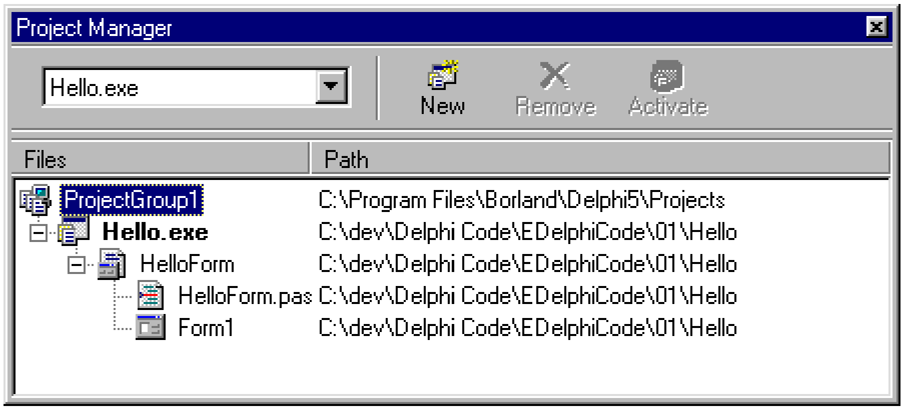
Once you've opened this file in Delphi, you can *edit* the textual description of the form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, this compilation will stop with an error message, and you'll need to correct the contents of your `DFM` file before you can reopen the form in the editor. For this reason, you shouldn't try to change the textual description of a form manually until you have a good knowledge of Delphi programming.

An expert programmer might choose to work on the text of a form for a number of reasons. For big projects, the textual description of the form is a powerful documenting tool, an important form of backup (in case someone plays with the form, you can understand what has gone wrong by comparing the two textual versions), and a good target for a version-control tool. For these reasons, Delphi also provides a DOS command-line tool, `CONVERT.EXE`, which can translate forms from the compiled version to the textual description and vice versa.

As we will see in the next chapter, the conversion is also applied when you cut or copy components from a form to the Clipboard.

The Project File

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR). This file is built automatically, and you seldom need to change it, particularly for small programs. If you do need to change the behavior of a project, there are basically two ways to do so: You can use the Delphi Project Manager and set some project options, or you can manually edit the project file directly.



Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi5\Projects
Hello.exe	C:\dev\Delphi Code\EDelphiCode\01\Hello
HelloForm	C:\dev\Delphi Code\EDelphiCode\01\Hello
HelloForm.pas	C:\dev\Delphi Code\EDelphiCode\01\Hello
Form1	C:\dev\Delphi Code\EDelphiCode\01\Hello

This project file is really a Pascal language source file, describing the overall structure of the program and its startup code:

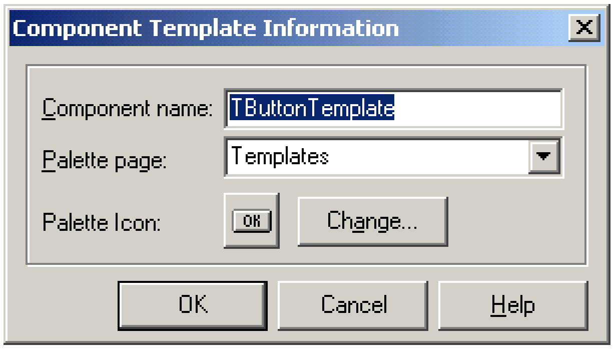
```
program Hello;

uses
  Forms,
  Hellof in 'HelloForm.PAS' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

You can see this file with the View | Project Source menu command. As an alternative, you can click on the Select unit button of the toolbar or issue the equivalent menu command, View | Units. When you use one of these commands, Delphi shows a dialog box with the list of the source files of the project. You can choose the project file (named Hello in the example), or any other file you are interested in seeing.

Using Component Templates

Suppose you want to create a brand new application, with a similar button and a similar event handler to the Hello program. It is possible to copy a component to the Clipboard, and then paste it into another form to create a perfect clone. However, doing so you copy only the properties of the component, and not the events associated with it.

Delphi allows you to copy one or more components, and install them as a new component template. This way, you also copy the code of the methods connected with the events of the component. Simply open the Hello example, or any other one, select the component you want to move to the template (or a group of components), and then select the Component | Create Component Template menu command. This opens the Component Template Information dialog box, shown below. Here you enter the name of the template, the page of the Component Palette where it should appear, and an icon.



Component name: TButtonTemplate

Palette page: Templates

What's Next

In this chapter, we created a simple program, added a button to it, and handled some basic events, such as a click with the mouse or a resize operation. We also saw how to name files, projects, forms, and components, and how this affects the source code of the program. We looked at the source code of the simple programs we've built, although some of you might not be fluent enough in Object Pascal to understand the details.

Before we can look into more complex examples, we need to explore the Delphi development environment. This is the topics of the next chapter. The examples in this chapter should have shown you that Delphi is really easy to use. Now we'll start to look at the complex mechanisms behind the scenes that make this all possible. You'll see that Delphi is a very powerful tool, even though you can use it to write programs easily and quickly.

Chapter 2: Highlights of the Delphi Environment

In a visual programming tool such as Delphi, the role of the environment is certainly important, at times even more important than the programming language used by its compiler or interpreter. This is a good reason to spend some time reading this chapter.

This chapter won't discuss all of the features of Delphi or list all of its menu commands. Instead, it will give you the overall picture and help you to explore some of the environment traits that are not obvious, while suggesting some tips that may help you. You'll find more information about specific commands and operations throughout the book.

Different Versions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single version of Delphi; there are three of them:

- The basic version (the "Standard" edition) is aimed at Delphi newcomers and casual programmers. It has all the features required to write programs in Delphi for Windows but (starting with Delphi 5) it has no support for any type of database programming.
- The second level (the "Professional" edition) is aimed at professional developers. It includes database support (with BDE and ODBC connectivity), limited Web support, and many more components.
- The full-blown Delphi (the "Enterprise" edition starting with Delphi 5, or the "Client/Server Suite" edition in previous versions) is aimed at developers building client/server applications. It includes also drivers for native Client/Server connection, full ADO support, MIDAS and Internet Express support, and (in Delphi) all of the new Snap technologies: BizSnap, WebSnap, and DataSnap.

Besides the different editions available, there are a number of ways to customize the Delphi environment. You can change the buttons of the toolbar, attach new commands to the Tools menu, hide some of the windows or elements, and resize and move all of them. In the screen illustrations throughout the book, I'll try to use a standard user interface (as it comes out of the box); however, I have my preferences, and I generally install many add-ons (written by third parties or by me), which might be reflected in some of the screen shots.

Asking for Help

Now we can really start our tour. The first element of the environment we'll explore is the Help system. There are basically two ways to invoke the Help system: select the proper command in the Help pull-down menu, or choose an element of the Delphi interface or a token in the source code and press F1.

> When you press F1, Delphi doesn't search for an exact match in the Help Search list. Instead, it tries to understand what you are asking. For example, if you press F1 when the text cursor is on the name of the Button1 component in the source code, the Delphi Help system automatically opens the description of the TButton class, since this is what you are probably looking for. This technique also works when you give the component a new name. Try naming the button Foo, then move the cursor to this word, press F1, and you'll still get the help for the TButton class. This means Delphi looks at the contextual meaning of the word for which you are asking help.

Note that there isn't just a single help file in Delphi. Most of the time, you'll invoke Delphi Help, but this file is complemented by an Object Pascal Help file, the Windows API Help, the Component Writer's Help, and many others (depending on your version of Delphi). These and other Help files have a common outline and a common search engine you can activate by pressing the Help Topics button while in the Help system. The Windows help engine dialog box that appears allows you to browse the contents of all of the help files in the group, search for a keyword in the index, or start the Find engine. The three capabilities are available in separate pages of the Help Topics dialog box.

You can find almost everything in the Help system, but you need to know what to search for. Usually this is obvious, but at times it is not. Spending some time just playing with the Help system will probably help you understand the structure of these files and learn how to find the information you need.

The Help files provide a lot of information, both for beginner and expert programmers, and they are especially valuable as a reference tool. They list all of the methods and properties for each component, the parameters of each method or function, and similar details, which are particularly important while you are writing code. Borland also distributes reference materials in the form of Adobe Acrobat files. These are electronic versions of the printed manuals that come in the Delphi box, so you can search them for a word, and you can also print the portions you are interested in (or even the whole file if you've got some spare paper).

> The first version of Delphi included some Interactive Tutors in addition to the Help system. If you've never used Delphi (and if you have Delphi 1 installed), you might consider running these Tutors. They will guide you through Delphi's basic features and help you understand some of the terminology of the environment. Unluckily they were soon discontinued by Borland.

Besides the Delphi Help files, there are many sources of collections of tips and suggestions for Delphi programmers. Borland web sites provides some FAQs (Frequently Asked Questions) and a collection of Technical Information short papers (TI). You can find updates of both at the Borland Community Web site (http://community.borland.com). Besides these official Borland documents, you'll find many more tips in Borland newsgroups (and also on mine). Obviously the Web is a great source of information about Delphi itself and third party products. You can find a collection of my favorite Delphi Web pages in the Links portion of my web site (http://www.marcocantu.com/links).

Delphi Menus and Commands

There are basically three ways to issue a command in the Delphi environment:

- Use the menu.

- Use the toolbar.
- Use one of the local menus activated by pressing the right mouse button.

The Delphi menus offer many commands. I won’t bore you with a detailed description of the menu structure. For this type of information, you can refer to the printed documentation or the Help file. In the following sections, I’ll present some suggestions on the use of some of the menu commands. Other suggestions will follow in the rest of the chapter.

The File Menu

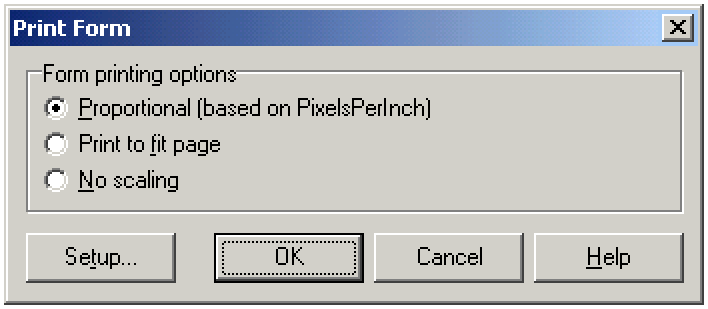
Our starting point is the File pull-down menu. The structure of this menu has kept changing from version to version of Delphi, with menu items for handling projects moving away, and specific commands to create new designer (for example data modules) coming and going. Still, this menu contains commands that operate on projects and commands that operate on source code files.

*[*** The File menu structure has changed in Delphi 6, with the File | New submenu, and the text here has not been updated accordingly]* Some commands can even be used to operate both on projects and on source code files. The commands related to projects are New, New Application, Open, Save Project As, Save All, Close All, Add to Project, and Remove from Project. Besides these, there is also a specific Project pull-down menu. The commands related to source code files are New, New Form, New Data Module, Open, Reopen, Save, Save As, Close, and Print. Most of these commands are very intuitive, but some require a little explanation.

> Use the Reopen menu command to open projects or source code files you have worked on recently.

The New command actually opens the New Items dialog box, also called the Object Repository. This dialog box can be used to invoke Delphi Wizards and to create items such as new applications, forms that inherit from existing forms, threads, DLLs, Delphi components, and ActiveX controls. I’ll cover the Object Repository’s rich set of features in the next chapter.

Another peculiar command is Print. If you are editing source code and select this command, the printer will output the text with syntax highlighting as an option. If you are working on a form and select Print from the File menu, the printer will produce the graphical representation of the form. This is certainly nice, but it can be confusing, particularly if you are working on other Delphi windows. Fortunately, two different print options dialog boxes are displayed, so that you can check that the operation is correct.



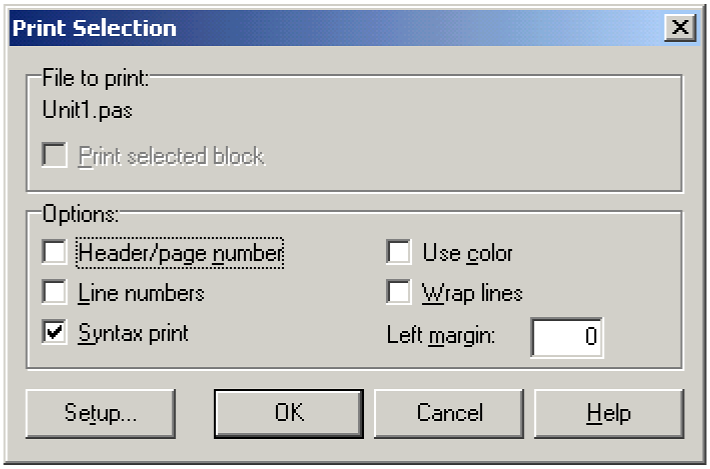
Print Form

Form printing options

● Proportional (based on PixelsPerInch)

○ Print to fit page

○ No scaling



File to print:
Unit1.pas
☐ Print selected block

Options:
☐ Header/page number ☐ Use color
☐ Line numbers ☐ Wrap lines
☑ Syntax print Left margin: 0

The Edit Menu

The Edit menu has some typical operations, such as Undo and Redo, and the Cut, Copy, and Paste commands, plus some specific commands for form or editor windows. The important thing to notice is that the standard features of the Edit menu (and the standard Ctrl+Z, Ctrl+X, Ctrl+C, and Ctrl+V keyboard shortcuts) work both with text and with form components. There are also some differences worth noting. For example, when you work with the editor, the first command of this pull-down menu is Undo; when you work with the form, it becomes Undelete. Unfortunately, the Form Designer has very limited Undo capabilities.

Of course, you can copy and paste some text in the editor, and you can also copy and paste components in one form, or from one form to another. You can even paste components to a different parent window of the same form, such as a panel or group box.

Besides using cut and paste commands, the Delphi editor allows you to move source code by selecting and dragging words, expressions, or lines. If you drag text while pressing the Ctrl key, it will be copied instead of moved.

Copying and Pasting Components

What you might not have noticed is that you can also copy components from the form to the editor and vice versa. Delphi places components in the Clipboard along with their textual description. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component.

For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi’s own source code editor or any word processor), you’ll get the following description:

```
object Button1: TButton
  Left = 56
  Top = 48
  Width = 161
  Height = 57
  TabOrder = 0
  Caption = 'Button1'
end
```

Now, if you change the name of the object, caption, or position, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object MyButton: TButton
  Left = 200
  Top = 200
  Width = 180
  Height = 60
  TabOrder = 0
  Caption = 'My Button'
  Font.Name = 'Arial'
end
```

Copying the above description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font. To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and change the position of the component so that it doesn't overlap a previous copy. You can see how Delphi modifies the properties of the component by copying it back to the editor. For example, this is what you get if you paste the text above in the form, and then copy it again into the editor:

```
object MyButton: TButton
  Left = 112
  Top = 128
  Width = 180
  Height = 60
  Caption = 'My Button'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 0
end
```

As you can see, some lines have been added automatically, to specify other properties of the font. Of course, if you write something completely wrong, such as this code:

```
object Button3: TButton
  Left = 100
  eight = 60
end
```

which has a spelling error (a missing 'H'), and try to paste it into a form, Delphi will show an error indicating what has gone wrong. You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

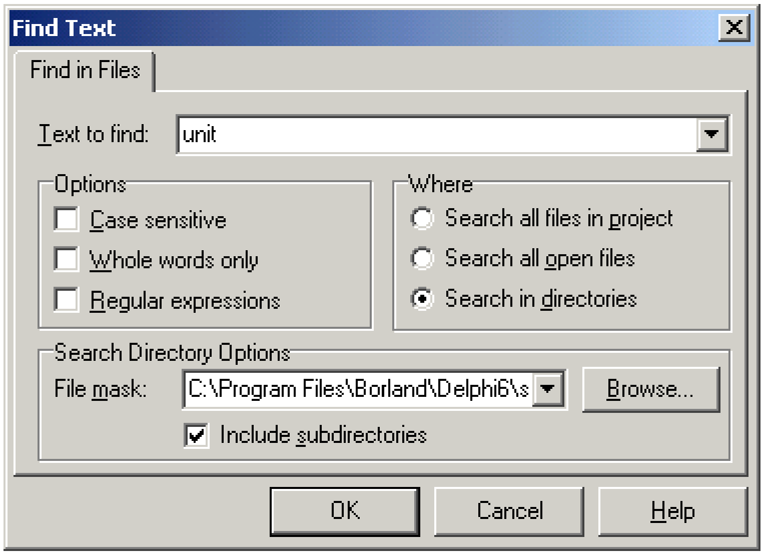
More Edit Commands

Along with the typical commands found on most Edit menus in Windows applications, Delphi includes a number of commands that are mostly related to forms. The specific operations for forms can also be accessed through the form shortcut menu (the local menu you can invoke with the right mouse button) and will be covered later in the chapter.

> One command not replicated in a form’s local menu is Lock Controls, which is very useful for avoiding an accidental change to the position of a component in a form. For example, you might try to double-click on a component and actually end up moving it. Since there is no Undo operation on forms, protecting from similar errors by locking the controls after the form has been designed can be really useful.

The Search Menu

The Search menu has some standard commands, too, such as Search and Replace, and the Find in Files command (you can see its dialog box here):



Find Text

Find in Files

Text to find: unit

Options

□ Case sensitive

□ Whole words only

□ Regular expressions

Where

○ Search all files in project

○ Search all open files

● Search in directories

Search Directory Options

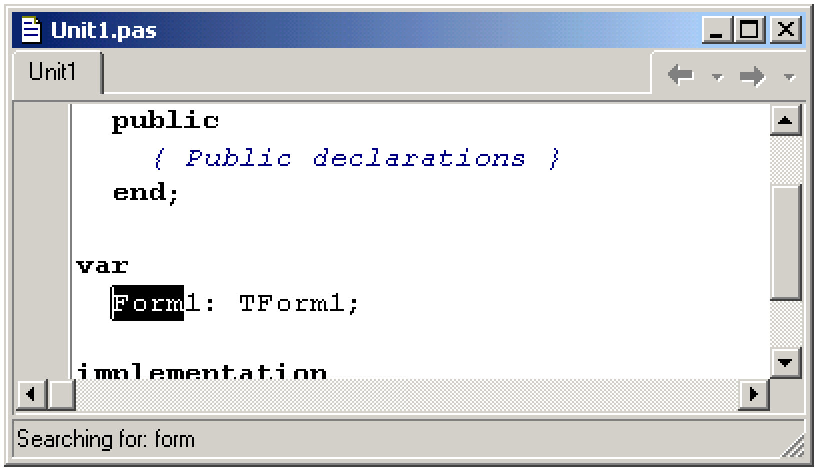
File mask: C:\Program Files\Borland\Delphi6\s

☑ Include subdirectories

The Find in Files command allows you to search for a string in all of the source code files of a project, all the open files, or all the files in a directory (optionally including its subdirectories), depending on the radio button you check. The result of the search will be displayed in the message area at the bottom of the editor window. You can select an entry to open the corresponding file and jump to the line containing the text.

> You can use the Find in Files command to search for component, class, and type definitions in the VCL source code (if your version of Delphi includes it). This is an easy way to get detailed information on a component, although using Help is generally faster and simpler.

Other commands are not so simple to understand. The Incremental Search command is one of them. When you select this command, instead of showing a dialog box where you enter the text you want to find, Delphi moves to the editor. There, you can type the text you want to search for directly in the editor message area, as you can see here:



```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

Searching for: form

When you type the first letter, the editor will move to the first word starting with that letter. (But if your search text isn't found, the letters you typed won't even be displayed in the editor message area.) If that is not the word you are looking for, just keep typing; the cursor will continue to jump as you add letters. Although this command might look strange at first, it is very effective and extremely fast, particularly if you are typing and invoke it with a shortcut key (Ctrl+E if you are using the standard editor shortcuts).

The Go to Line Number command is quite intuitive. The Find Error command might seem strange at first. It is used to find a particular run-time error, not to search for a compiler error. When you are running a stand-alone program and you hit a very bad error, Delphi displays an internal address number (that is, the *logical* address of the compiled code). You can enter this value in the Find Error dialog box to have Delphi recompile the program, looking for the specific address. When it finds the address, Delphi shows the corresponding source code line. Often, however, the error is not in one of the lines of your code, but in a line of library or system code; in this (quite frequent) case the Find Error command cannot locate the offending line.

The last command on the Search menu, Browse Symbol, invokes the Object Browser, a tool you can use to explore all the symbols defined in a compiled program. To understand the output of the Object Browser, you need a good understanding of the Object Pascal language and of the Visual Component Library (VCL).

The View Menu

The View pull-down menu combines the features you usually find in View and Window menus. There is no Window menu, because the Delphi environment is not an MDI application. Most of the View commands can be used to display one of the windows of the Delphi environment, such as Project Manager, the Breakpoints list, or the Components command. Some of these windows are used during debugging; others when you are writing code. Most of these windows will be described later in this chapter.

> It is possible to add a new item to the View menu, CPU Window, which can be used during debugging to view the Assembler code generated by the Delphi compiler, execute it step by step, and view the status of the CPU registers.

The commands on the second part of the View menu are important, which is why they are also available on the default toolbar. The Toggle Form/Unit (or F12) command is used to move between the form you are working on and its source code. If you use a source code window big enough to hold a reasonable amount of text, you'll use this command often. As an alternative, you can place the two windows (the editor and the form) so that a portion of the one below is always visible. With this arrangement, you can click on it with the mouse to move it to the front.

The New Edit Window command opens a second edit window. It is the only way to view two files side by side in Delphi, since the editor uses tabs to show the multiple files you can load. Once you have duplicated the edit window, you can make each one hold a different set of files, or view two portions of the same file.

The last two commands on the View menu can be used to hide the toolbar or the Components palette, although this is a good way to make Delphi look silly and uncomfortable. Working on forms without the Components palette is certainly not easy. If you remove both the toolbar and the Components palette, the Delphi main window is reduced to a bare menu.

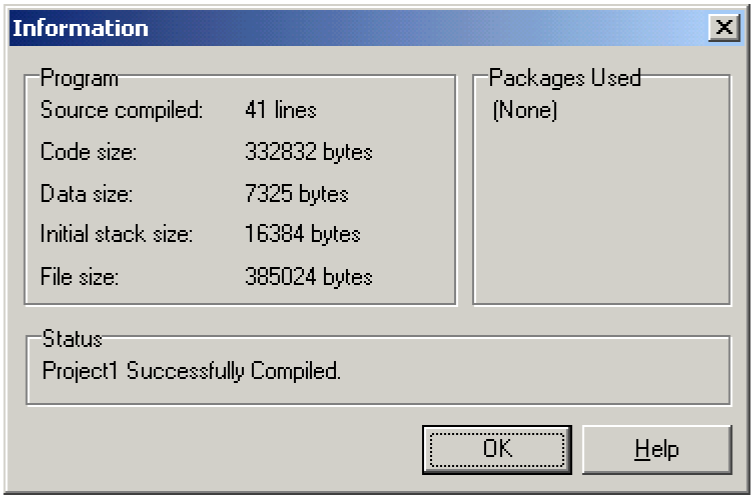
The Project Menu

The next pull-down menu, Project, has commands to manage a project and compile it. Add to Project and Remove from Project are used to add forms or Pascal source code files to a program and to remove them from a project.

> The Enterprise version of Delphi includes two more commands, Web Deploy Options and Web Deploy, which are not available in the other editions. These features are related with ActiveX and ActiveForms, some (now obsolete) Microsoft web technologies.

The Compile command builds or updates the application executable file, checking which source files have changed and recompiling them when needed. With Build All, you can ask Delphi to compile every source file of the project, even if it has not been changed since the last compilation. If you just want to know whether the syntax of the code you've written is correct, but you do not want to build the program, you can use the Syntax Check command.

The next Project command, Information, displays some details about the last compilation you've made. The following figure shows the information related to the compilation of the a short program:



Source compiled: 41 lines
Code size: 332832 bytes
Data size: 7325 bytes
Initial stack size: 16384 bytes
File size: 385024 bytes

(None)

Project1 Successfully Compiled.

> The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU (Delphi Compiled Unit).

At the end of the Project menu comes the Options menu, used to set compiler and linker options, application object options, and so on. When you change the project options, you can check the Default box to indicate that the same set of options should be used for new projects. We will discuss project options again in this chapter and then throughout the book in the context of related topics.

The Run Menu

The Run menu could have been named Debug as well. Most of its commands are related to debugging, including the Run command itself. When you run a program within the Delphi environment, you execute it under the integrated debugger (unless you disable the corresponding Environment option). The Run command and the corresponding toolbar icon are among the most commonly used commands, since Delphi automatically recompiles a program before running it — at least if the source code has changed. Simply hit F9 as a shortcut to compile and run a program.

The next command, Parameters, can be used to specify parameters to be passed on the command line to the program you are going to run, and to provide the name of an executable file when you want to debug a DLL (DLL debugging is another new Delphi 3 feature). The remaining commands are all used during debugging, to execute the program step by step, set breakpoints, inspect the values of variables and objects, and so on. Some of these debugging commands are also available directly in the editor local menu.

Delphi 3 has a couple of new commands related to ActiveX development (and not found in the “Standard” edition). The Register ActiveX Server and Unregister ActiveX Server menu commands basically add or remove the Windows Registry information about the ActiveX control defined by the current project.

The Component Menu

The commands of the Component menu can be used to write components, add them to a package, or to install packages in Delphi. The New Component command invokes the simple Component Wizard. The three installation commands, Install Component, Import ActiveX Library, and Install Packages, can be used to add to the environment new Delphi components, packages, or ActiveX controls. Executing any of these commands adds the new components to the specified package and to the Components palette.

Component Templates

We briefly used the Create Component Template menu item in the last chapter. When you issue this command after selecting one or more components in a form, it opens a dialog box where you specify the name of the new component template, a page on the Palette, and an icon. By default, the template name is the name of the first component you’ve selected followed by the word *template*. The default template icon is the icon of the first component you’ve selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Components palette (when Delphi displays the fly-by hint).

All the information about component templates is stored in a single file, DELPHI32.DCT, but there is apparently no way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand new form, edit it, and install it again as a component template *using the same name*. This way you can override the previous definition.

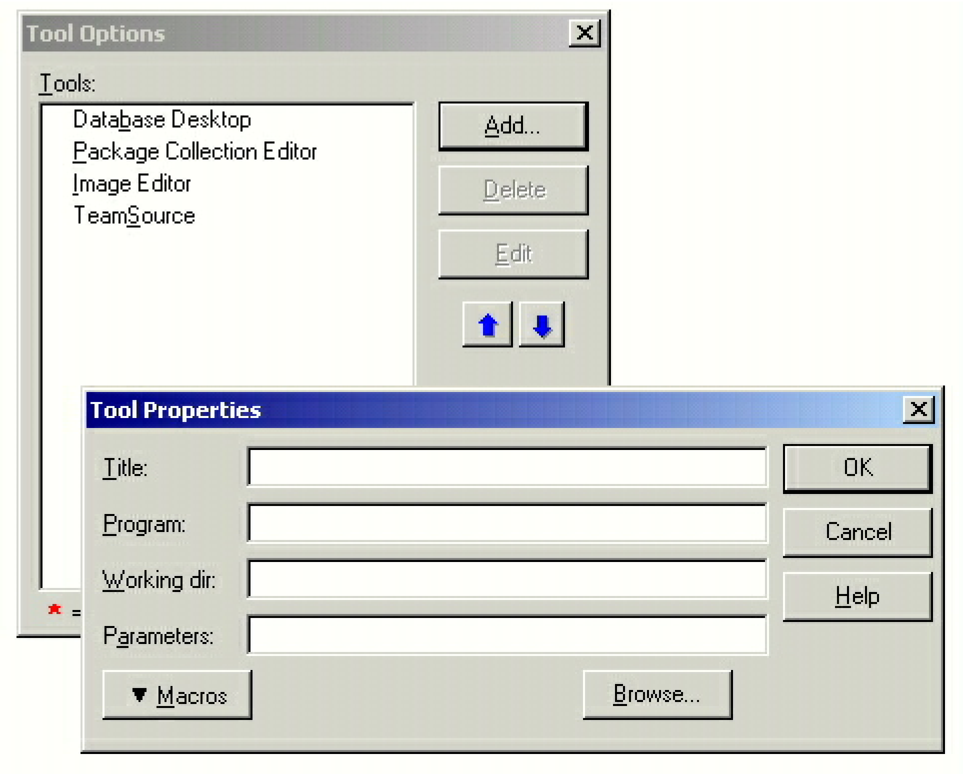
The Database Menu

The Database menu collects the Delphi database-related tools, such as the Database Form Wizard and the Database Explorer. The Enterprise edition has the SQL Explorer instead of the Database Explorer (although the menu items is invariably called Database | Explore) and a menu item to start the SQL Monitor.

The Tools Menu

The Tools menu simply lists a number of external programs and tools, just to make it easier to run them. You can use the Tools command to configure and add new external tools to the pull-down. Besides simply running a program, you can pass some parameters to it. Simple parameter lists can be passed directly on the command line, while complex ones can be built by clicking the Macros button in the lower part of the Tool Properties dialog box.

The Tools menu also includes a command to configure the Repository (discussed in the next chapter) and the Options command to configure the whole Delphi development environment. The Environment Options dialog box has many pages related to generic environment settings (the Preferences page), packages and library settings, many editor options (in the pages Editor, Display, and Colors), a page to configure the Components Palette, one for the object Browser, and one of the new Code Insight technology. I'll discuss many of these options when covering related features. You can see the dialog used to customize the Tools menu below.



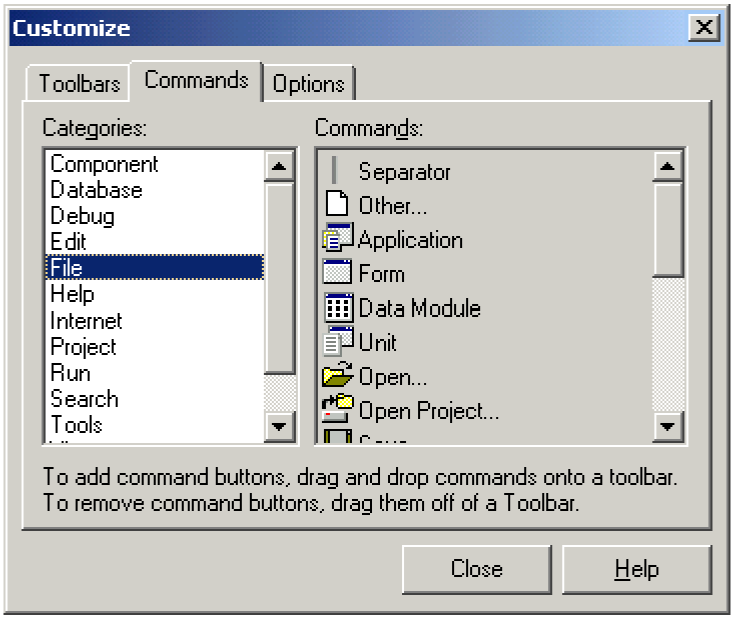
The Help Menu

The Help menu can be used to get information about Delphi (Help | Help Topics) and also to display the Delphi About box. In this window, you can hold down the Alt key and type the letters VERSION to see the Delphi version and build number. Using other key combinations (as mentioned in the acknowledgments at the beginning of the book) you can see a list of the people involved in building Delphi. The Help menu was often populated by third-party Delphi Wizards (before the new ToolsApi made it harder to place wizards in what was their default location).

The Delphi Toolbar

After you have used Delphi for a while, you'll realize that you use only a small subset of the available commands frequently. Some of these commands are probably already on the toolbar (Borland's name for a toolbar); some are not. If the commands you use a lot are not there, it's time to customize the toolbar so that it really helps you to use Delphi more efficiently.

> An alternative to using the toolbar is to use shortcut keys. Although you must remember some key combinations to use them, shortcut keys let you invoke commands very quickly, particularly when you are writing code and your fingers are already on the keyboard.



To add command buttons, drag and drop commands onto a toolbar.
To remove command buttons, drag them off of a Toolbar.

You can easily resize the toolbar by dragging the thick line between it and the Components Palette. But the most important operations you can do with the toolbar are adding, removing, or replacing the icons using the Configure command of the toolbar local menu (simply press the right mouse button over it). This operation invokes the toolbar Editor (see above), one of the Delphi tools with the best user interface, at least in my opinion.

To add an icon to the toolbar, you simply need to find it under the proper category (corresponding to a pull-down menu), and drag it to the bar. In the same way, you can drag an icon away from the toolbar or simply move it to another location. During these operations, you can easily leave some space between groups of icons, to make them easier to remember and select.

The Local Menus

Although Delphi has a good number of menu items, not all of the commands are available though the pull-down menus. At times, you need to use local menus for specific window areas. To activate a local menu, right-click over a window, or press Alt+F10. Even if you have other alternatives, using a local menu is usually faster because you don't need to move the mouse up to the menu bar and select two levels of menus. It's also often easier, since all the local menu commands are related to the current window. Almost every window in Delphi (with the exclusion of dialog boxes) has its own local menu with related commands. I really suggest you get used to right-clicking on windows, because this is not only important in Delphi, but also has become a standard for most applications in Windows. Get used to it, and add local menu to the applications you build with Delphi, too.

Working with the Form Designer

Designing forms is the core of visual development in the Delphi environment. Every component you place on a form and every property you set is stored in a file describing the form (a DFM file) and has some effect on the source code associated with the form (the PAS file).

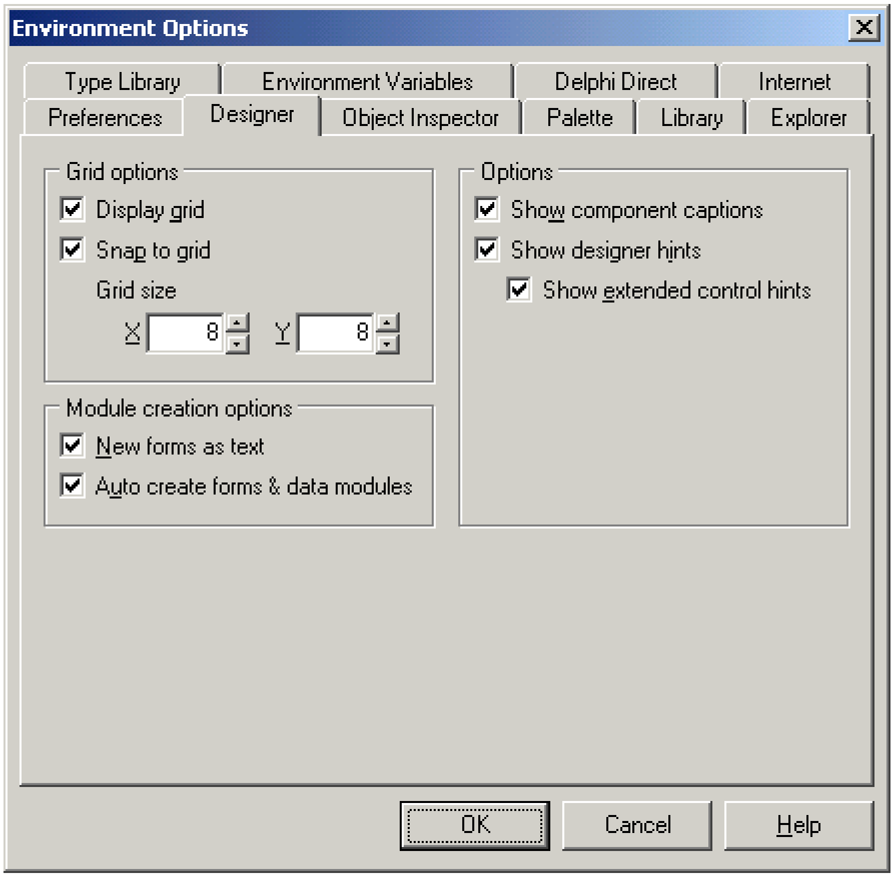
When you start a new, blank project, Delphi creates an empty form, and you can start working with it. You can also start with an existing form (using the various templates available), or add new forms to a project. A project (an application) can have any number of forms. Every time you work with a form at design-time, you are actually using Delphi's Form Designer. When you are working with a form, you can operate on its properties, on the properties of one of its components, or on those of several components at a time. To select the form or a component, you can simply click on it or use the Object Selector (the combo box in the Object Inspector), where you can always see the name and type of the selected item. You can select more than one component by Shift-clicking on the components, or by dragging a selection rectangle around the components on the form.

> Even when a component covers the whole surface of the form, you can still select the form with the mouse. Just press and hold Shift while you click on the selected component. This will deselect the component and select the form by default. Using the keyboard, you can press Esc to select the parent of the current component.

While you are working on a form, the local menu has a number of useful features (some of which are also available in the Edit menu). You can use the Bring to Front and Send to Back commands to change the relative position of components of the same kind (you can never bring a graphical component in front of a component based on a window). In an inherited form, you can use the command Revert to Inherited to restore the properties of the selected component to the values of the parent form.

When you have selected more than one component, you can align or size them. Most of the options in the Alignment dialog box are also available in the Alignment palette (accessible through the View | Alignment Palette menu command). You can also open the Tab Order and Creation Order dialog boxes to set the tab order of the visual controls and the creation order of the non-visual controls. You can use the Add to Repository command to add the form you are working on to a list of forms available for use in other projects. Finally, you can use the View as Text command to close the form and open its textual description in the editor. A corresponding command in the editor local menu (View as Form) will reverse the situation.

Along with specific local menu commands, you can set some form options by using the Tools | Options command and choosing the Preferences page (up to Delphi 5) or the Designer page (from Delphi 6). This latter page is shown here:



Grid options

☑ Display grid

☑ Snap to grid

Grid size

X 8 Y 8

Module creation options

☑ New forms as text

☑ Auto create forms & data modules

Options

☑ Show component captions

☑ Show designer hints

☑ Show extended control hints

The options related to forms refer to grid activation and size. The grid makes it easier to place components exactly where you want them on the form by "snapping" them to fixed positions and sizes. Without a grid, it is difficult to align two components manually (using the mouse).

There are two alternatives to using the mouse to set the position of a component: you can either set values for the Left and Top properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position. (The Snap to Grid option works only for mouse operations.) Similarly, by pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. If you press Shift+Ctrl+an arrow key, instead, the component will be moved only at grid intervals.

Along with the commands described so far, a form's local menu offers other commands when particular components are selected. In some cases, these menu commands correspond to component properties; others contain particularly useful commands. Table 2.1 lists the commands added to the local menu of a form when some of the components are selected (the TeeChart, Quick Report and Decision Cube components add too many commands to list here). Notice that in some cases these actions are also the default action of the component, the one automatically activated when you double-click on it in the Form Designer.

Table 2.1: Local menu commands added when specific components are selected: [*** not updated for recent versions of Delphi]

Menu Command	**Components**
Menu Designer	MainMenu, PopupMenu

Query Builder	Query (if the Visual Query Builder is available)
Fields Editor	Table, Query, StoredProc, ClientDataSet
Explore	Table, Query, StoredProc, Database
Define Parameters	Query, StoredProc
Database Editor	Database
Assign Local Data	ClientDataSet
UpdateSQL Editor	UpdateSQL
Execute	BatchMove
Columns Editor	DBGrid
Edit Report	Report
ImageList Editor	ImageList
New Page	Page Control
Next/Previous Page	Page Control, NoteBook, TabbedNotebbok
Next/Previous Frame	Animate
Insert Object	OleContainer
New Button	ToolBar
New Separator	
Properties, About	All the ActiveX controls
Action Editor	WebDispatcher
ResponseEditor	QueryTableProducer, DataSetTableProducer

The Component Palette

When you want to add a new component to the form you are working on, you can click on a component in one of the pages of the Component palette, and then click on the form to place the new component. On the form, you can press the left mouse button and drag the mouse to set the position and size of the component at once, or just click to let Delphi use a default size.

Each page of the palette has a number of components; each component has an icon and a name, which appears as a “fly-by” hint (just move the mouse on the icon and wait for a second). The hints show the official names of components, which I’ll use in this book. They are drawn from the names of the classes defining the component, without the initial *T* (for example, if the class is TButton, the name is *Button*).

> If you need to place a number of components of the same kind into a form, shift-click on that component in the palette. Then, every time you click on the form, Delphi adds a new component of that kind. To stop this operation, simply click on the standard selector (the arrow icon) on the left side of the Component palette.

If you are temporarily using a mouse-less computer, you can add a component by using the View | Components List command. Select a component in the resulting list or type its name in the edit window, and then click on the Add to Form button.

Of course, you can completely rearrange the components in the various pages of the palette, adding new elements or just moving them from page to page: select Tools | Options and move to the Palette page. In this page of the dialog box, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page. It’s not a good idea to move components on the palette too often. If you do, you’ll probably waste time trying to locate them afterward.

> When you have too many pages in the Component palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: rename the pages with shorter names, so that all the pages will fit on the screen. Obvious...once you've thought about it.

The Object Inspector

When you are designing a form, you use the Object Inspector to set values of component or form properties. Its window lists the properties (or events) of the selected element and their values in two resizable columns. An Object Selector at the top of the Object Inspector indicates the current component and its data type; and you can use it to change the current selection. The Object Inspector doesn't list all of the properties of a component. It includes only the properties that can be set at design-time. As mentioned in Chapter 1, other properties are accessible only at run-time. To know about all the different properties of a component, refer to the Help files.

The right column of the Object Inspector allows only the editing appropriate for the data type of the property. Depending on the property, you will be able to insert a string or a number, choose from a list of options (indicated by an arrow), or invoke a specific editor (indicated by an ellipsis button). When a property allows only two values, such as True and False, you can toggle the value by double-clicking on it. If there are many values available, a double-click will select the next one in the list. If you double-click a number of times, all the values of the list will appear, but it is easier to select a multiple-choice value using the small combo box. For some properties, such as Color, you can enter a value, select an element from the list, or invoke a specific editor! Other properties, such as Font, can be customized either by expanding their sub-properties (indicated by a plus or minus sign next to the name) or by invoking an editor. In other cases, such as with string lists, the special editors are the only way to change a property.

The sub-property mechanism is available with sets and with classes. When you expand sub-properties, each of them has its own behavior in the Object Inspector, again depending on its data type.

You will use the Object Inspector often. It should always be visible when you are editing a form, but it can also be useful to look at the names of components and properties while you are writing code. For this reason, the Object Inspector's local menu has a Stay on Top command, which keeps the Object Inspector window in front of the Form Designer and the editor.

> In Delphi 5 and 6 the features of the Object Inspector have been largely extended, with grouping and hiding or little-used properties, interface references, showing of read-only properties, and many other features too complex to discuss here.

The Alignment Palette

The last tool related to form design is the Alignment palette. You can open this palette with the View menu's Alignment Palette command. As an alternative, you can choose the components you want to align, and then issue the Align command from the local menu of the form.

The Alignment palette features a number of commands to position the various controls, center them, space them equally, and so on. To see the effect of each button, simply move the mouse over the window and look at the fly-by hints. When I'm designing complex forms, I position the Alignment palette on the far side of the screen and make sure it always stays in sight by using the Stay on Top command of its local menu.

Writing Code in the Editor

Once you have designed a form in Delphi, you usually need to write some code to respond to some of its events, as we did in Chapter 1. Every time you work on an event, Delphi opens the editor with the source file related to the form. You can easily jump back and forth between the Form Designer and the source code editor by clicking the Toggle Form Unit button on the toolbar, by clicking on the corresponding window, or by pressing the F12 function key.

The Delphi editor allows you to work on several source code files at once, using a "notebook with tabs" metaphor. Each page of the notebook corresponds to a different file. You can work on units related to forms, independent units of Pascal code, and project files; open the form description files in textual format; and even work on plain text files. You can jump from a page of the editor to the next by pressing the Ctrl+Tab keys (or Shift+Ctrl+Tab to move in the opposite direction).

When you work with the editor, you should probably expand its window so that you can see as many full lines of code as possible. A good approach is to size the editor so that it and the Object Inspector are the only windows that appear on the screen when you are writing code. By having the Object Inspector visible you can immediately see the names of the design-time properties of the components.

There are a number of environment options that affect the editor, mostly located in the Editor Options, Editor Display, and Editor Colors pages of the Environment Options dialog box. In the Preferences page, you can set the editor's Autosave feature. Saving the source code files each time you run the program can save the day when your program happens to crash the whole system (something not so rare as you might think). The other three pages of editor options can be used to set the default editor settings like keystroke mappings, syntax highlighting features, and font. Most of these options are fairly simple to understand.

The local menu of the edit window has some commands for debugging and others related to the editor itself, such as those to close the current page, open the file or unit under the cursor, view or hide the message pane below the window, and invoke the editor options discussed before.

Using Editor Bookmarks

The Delphi editor also lets you set line bookmarks. When you are on a line of the editor, you can press Ctrl+Shift plus a number key from 0 to 9 to set a new bookmark, which then appears in the small gutter margin of the editor. Then you can use the Ctrl key plus the number key to jump back at that line of the editor. Pressing the Ctrl+Shift+*number* toggles the status of the bookmark, so you can use this combination again to remove it.

Bookmarks are quite useful when you have a long file and you are editing multiple methods at the same time, or to jump from the class definition to the definition of a method of the class.

Bookmarks have limitation that make them hardly usable. If you set again a given bookmark, the editor moves it. This might seem reasonable, but is actually a problem: If you create a new bookmark and happen to use the number of an existing one, by error, the older bookmark will be removed. Another odd behavior is that you can add multiple bookmarks on the same line, but you'll only see the glyph of one of them. The real problem, however, is that bookmarks are not saved along with the file, nor restored when you reopen it. So they can be used only for a single editing session.

Code Insight

Delphi editor has several features collectively known as Code Insight. The basic idea of this technology is to make it easier for both newcomers and experienced programmers to write code. There are four capabilities that Borland calls Code Insight:

- The Code Completion Wizard allows you to choose the property or method of an object simply by looking it up on a list, or by typing its initial letters. It also allows you to look for a proper value in an assignment statement.
- The Code Templates Wizard allows you to insert one of the predefined code templates, such as a complex statement with an inner begin-end block. You can also easily define new templates.
- The Code Parameter Wizard displays, in a hint or ToolTip window, the data type of a function's or method's parameters while you are typing them.
- The ToolTip Expression Evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression under the mouse cursor.

I'll cover the ToolTip Expression Evaluation later in this chapter, while introducing debugging features; in the next three sections I'll give you some more details on the other three Code Insight capabilities. You can enable and disable (or configure) each of these wizards in the Code Insight page of the Environment Options dialog box.

Code Completion

There are two ways to activate this Wizard. You can simply type the name of an object, such as Button1, then add the dot, and wait:

```
Button1.
```

Delphi will display a list of valid properties and methods you can apply to the object. The time you have to wait before the list is displayed depends on the Code Completion Delay option, which you can configure in the Code Insight page.

As an alternative you can type the initial portion of the property or method name, as in Button1.Ca, and then press Ctrl+SpaceBar to get the list immediately, but this time, the Wizard will try to guess which property or method you were looking for by looking at the characters you typed. You can also use this key combination in an assignment statement. If you type:

```
x :=
```

and then press Ctrl+SpaceBar, Delphi will show you a list of possible objects, variables, or constants you can use at this point in the program (that is, in the current scope).

Notice that Delphi determines the elements to show in this list dynamically, by constantly parsing the code you write in the background. So if you add a new variable to a unit, it will show up in the list.

Code Templates

Unlike the Code Completion Wizard, the Code Templates Wizard must be activated manually. You can do this by typing Ctrl+J to show a list of all of the templates.

More often, you'll first type a keyword, such as if or array, and then press Ctrl+J, to activate only the templates starting with those letters. For some keywords Borland has defined multiple templates, all starting with the keyword name (such as *ifA* and *ifB*). So if you press the keyword and then Ctrl+J, you'll get all the templates related to the keyword.

You can also use Code Templates Wizard simply to give a name to a common expression. For example, if you use the MessageDlg function often, you might want to enter a new Code Template called *mess*, type a description, and add then the following text:

```
MessageDlg ('|',
  mtInformation, [mbOK], 0);
```

Now every time you need to create a message dialog box, you simply type *mess* and then Ctrl+J, and you get the full text. The vertical line (or pipe) character indicates the position in the source code where Delphi will move the cursor after pasting the text. You should choose the position where you want to start typing to edit the code generated by the template.

As this example demonstrates, Code Templates have no direct correspondence to language keywords, but are a more general mechanism. Code Templates are saved in the DELPHI32.DCI file, so it should be possible to copy this file to make your templates available on different machines. There seems to be no easy way to merge two Code Templates files, and there are no third-party tools yet to add more templates to a machine. Those enhancements will need to wait until Borland documents the internal structure of the .DCI file.

Code Parameter

The third Code Insight technology I'll discuss here is Code Parameters, the one I was really hoping for and probably like best. Previously when I had to call an unfamiliar function I used to type the name, and then press F1 to jump to the Help system and see its parameters. Now I can simply type the function name, type the open (left) parenthesis, and the parameter names and types appear immediately on a fly-by hint window.

Notice in this figure that the first parameter appear in boldface type. After you type the first parameter and a comma, the second parameter will be set in bold, the same with the third, and so on. This is very useful for functions with many parameters, like some functions of the Window API. Try typing *CreateWindow(* and you'll understand what I mean.

Again, the Code Parameters Wizard works by parsing your code in the background. So if you write the following procedure at the beginning of the implementation section of a unit:

```
implementation

procedure ShowInt (X: Integer);
begin
  MessageDlg (IntToStr (X),
    mtInformation, [mbOK], 0);
end;
```

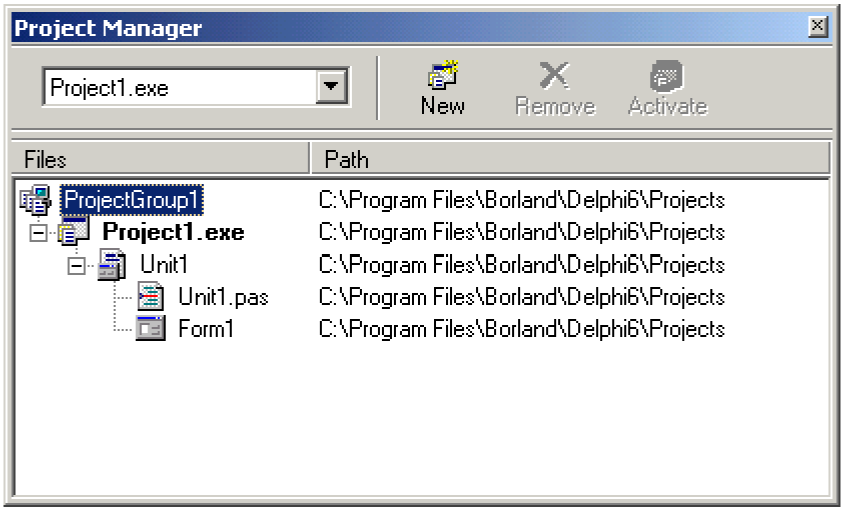
you'll have full information about its parameters when you type *ShowInt(* later.

Managing Projects

In Delphi you also need to know how to manage project files. In Chapter 1, we saw that you can open a project file in the editor and edit the file. However, there are simpler ways to change some of the features of a project. For example, you can use the Project Manager window and Project Options.

The Project Manager

When a project is loaded, you can choose the View | Project Manager command to open a project window. The window lists all of the forms and units that make up the current project. The Project Manager's local menu allows you to perform a number of operations on the project, such as adding new or existing files, removing files, viewing a source code file or a form, and adding the project to the repository. Most of these commands are also available in the toolbar of this window.



Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi6\Projects
Project1.exe	C:\Program Files\Borland\Delphi6\Projects
Unit1	C:\Program Files\Borland\Delphi6\Projects
Unit1.pas	C:\Program Files\Borland\Delphi6\Projects
Form1	C:\Program Files\Borland\Delphi6\Projects

Setting Project Options

From the Project Manager (or from the Project menu), you can invoke the Project Options dialog. The first page of Project Options, named Forms, lists the forms that should be created automatically at program startup (the default behavior) and the forms that are created manually by the program. You can easily move a form from one list to the other. The next page, Application, is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

> There are two ways to set compiler options. One is to use the Compiler page of the Project Options, the other is to set or remove individual options in the source code with the {$X+} or {$X-} commands, where X is the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source code file, or even for just a few lines of code.

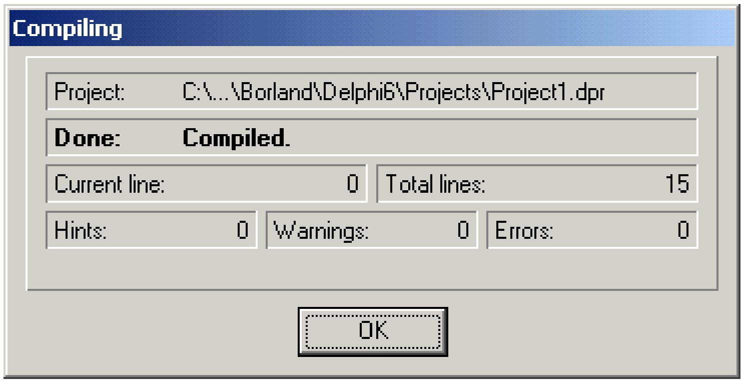
All of the Project Options are saved automatically with the project, but in a separate file with a DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options.

Compiling a Project

There are several ways to compile a project. If you run it (by pressing F9 or clicking on the toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed. If you select Compile | Build All, instead, every file is compiled, even if it has not changed. This second command is seldom used, since Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options. In this case you have to use the Build All command to put the new options into effect.

The project lists the source code files that are part of it, and any related forms. This list is visible both in the project source and in the Project Manager, and is used to compile or rebuild a project. First, each source code file is turned into a Delphi compiled unit, a file with the same name as the Pascal source file and the DCU extension. For example, Unit1.pas is compiled into Unit1.dcu.

When the source code of the project itself is compiled, the compiled units that constitute the project are merged (or linked) into the executable file, together with code from the VCL library. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option. You'll find this option on the Preferences page of the Environment Options dialog box, under the Compiling heading. Although this slows down the compilation a little, the Compile window lets you see which source files are compiled each time (unless your computer is too fast; Delphi might compile several files per second on a fast PC).



Project: C:\...\Borland\Delphi6\Projects\Project1.dpr

Done: Compiled.

Current line: 0
Total lines: 15

Hints: 0
Warnings: 0
Errors: 0

Exploring a Compiled Program

Delphi provides a number of tools you can use to explore a compiled program, including the debugger and the Object Browser.

The Integrated Debugger

Delphi has an integrated debugger with a huge number of features. However, Borland also sells a more powerful stand-alone debugger, called Turbo Debugger. For nearly all of your debugging tasks, the integrated debugger works well enough, particularly if you activate the CPU view window. The stand-alone Turbo Debugger might be useful in a few special cases.

You don’t need to do much to use the integrated debugger. In fact, each time you run a program from Delphi, it is executed by default in the debugger. This means that you can set a breakpoint to stop the program when it reaches a specific line of code. For example, open the Hello2 example we created in Chapter 1 and double-click on the button in the form to jump to the related code. Now set a breakpoint by clicking in the editor gutter margin, by choosing the Toggle Breakpoint command of the editor local menu, or by pressing F5.

The editor will highlight the line where you’ve placed the breakpoint, showing it in a different color. Now you can run the program as usual, but each time you press the button, the debugger will halt the program, showing you the corresponding line of code. You can execute this and the following lines one at a time (that is, step-by-step), look at the code of the functions called by the code, or continue running the program.

When a program is stopped, you can inspect its status in detail. Although there are many ways to inspect a value, the simplest approach is the ToolTip Expressions Evaluation. Simply move the mouse over the name of any variable, and you’ll see its current value in a small hint window.

At times the ToolTip Expressions Evaluation seems not to work. This may happen if the optimizing compiler has removed some sections of generated code and placed variables in CPU registers. If you disable the compiler optimizations, you’ll get more ToolTips.

The Object Browser

Once you have compiled a program, you can run the Object Browser (available with the View | Browser menu command) to explore it, even if you are not running or debugging it. This tool allows you to see all of the classes defined by the program (or by the units used directly and indirectly by the program), all the global names and variables, and so on. For every class, the Object Browser shows the list of properties, methods, and variables — both local and inherited, private and public. The information displayed in the Object Browser may not mean much if you’re still not familiar with the Object Pascal language used by Delphi.

Additional Delphi Tools

Delphi provides many other programming tools. For example, the Menu Designer is a visual tool used to create the structure of a menu. There are also the various Wizards, used to generate the structure of an application or a new form. Other tools are stand-alone applications related to the development of a Windows application, such as the Image Editor and WinSight, a “spy” program that lets you see the Windows message flow.

There are several external database tools, such as the Database Desktop and the Database Explorer (some of which are not available in the lower-level editions of Delphi). A programmer can use other third-party tools to cover weak areas of Delphi. For example, you can use a full-blown resource editor (such as Borland’s Resource Workshop), or a tool to generate Help files more easily.

You can also install and use many additional Delphi add-on tools from third-party vendors. You’ll find demo versions of some of these tools on the companion CD, but there are many others available. Some of the add-on tools really complement Delphi nicely, and make you more productive.

The Files Produced by the System

As you have seen, Delphi produces a number of files for each project, and you should know what they are and how they are named. There are basically two elements that have an impact on how files are named: the names you give to a project and its forms, and the predefined file extensions used by Delphi for the files you write and those generated by the system.

The great advantage of Delphi over other visual programming environments is that most of the source code files are plain ASCII text files. We explored Pascal source code, project code, and form description files at the end of Chapter 1. Now let’s take a minute to look at the structure of options and desktop files. Both types of files use a structure similar to Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of the HELLO.DOF file of the Hello2 example:

```
[Compiler]
A=1
B=0
...
[Linker]
MapFile=0
MinStackSize=16384
MaxStackSize=1048576
...
[Directories]
OutputDir=
SearchPath=
```

> In Delphi the option files use the DOF extension, in Kylix the KOF extension (in Delphi 1 they used the OPT extension. These files have different contents and are not fully compatible.

The initial part of this file, which I've omitted, is a long list of compiler options. The same structure is used by the desktop files, which are usually much longer. It is worth looking at what is stored in these files to understand their role. In short, a desktop file (.DSK) lists Delphi windows, indicating their position and status. For example, this is the description of the main window:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

These are some of the sections related to other windows:

```
[ProjectManager]
[AlignmentPalette]
[PropertyInspector]
[Modules]
[formxxx]
[EditWindowxxx]
[Viewxxx]
```

Besides environment options and window positions, the desktop file contains a number of history lists (lists of files of a certain kind), and an indication of the current breakpoints, watches, active modules, closed modules, and forms.

What's Next

This chapter presented an overview of the Delphi programming environment, including a number of tips and suggestions. Getting used to a new programming environment takes some time, particularly if it is a complex one. I could have devoted this entire book to detailing the Delphi programming environment, but I hope you'll agree that describing how to actually write programs is more useful and interesting.

A good way to learn about the Delphi environment is to use the Help system, where you can look up information about the environment elements, windows, and commands. Spend some time just browsing through the Help files. Of course, the best way to learn how the Delphi environment works is to use it to write programs. That's what Delphi is about. Now we can move on to an important feature of the Delphi environment we have only mentioned: the Object Repository and the Wizards.

CHAPTER THREE: THE OBJECT REPOSITORY AND THE DELPHI WIZARDS

- Delphi's Object Repository
- Reusing existing applications and forms
- The Database Form Wizard
- Other Delphi Wizards
- Configuring the Object Repository

When you start working on a new application (or simply a new form), you have two choices. You can start from scratch with a blank application or form, or you can choose a predefined model from the Object Repository. If you decide to pick an existing model from the Object Repository, you have even more alternatives. You can make a copy of an existing item (called a template in Delphi 1), you can inherit from an existing item, or you can use one of the available Wizards.

A *Wizard* is a code generator. (Borland used to call them *Experts*, but in Delphi 3 the official name has been changed to Wizards, following Microsoft tradition.) Wizards ask you a number of questions, and use your answers to create some basic code, following predefined rules. You then start working on a project or a form that already has some code and components. Usually, the code generated by these tools can be compiled immediately, and it makes up the basic structure on which you build your program or form.

The purpose of this short chapter is simply to introduce you to the Object Repository and the Delphi Wizards, and to show you how easy they are to use. We won't study the code they generate, since that will be the topic of many examples in the book. From a programming standpoint, the Wizards are really useful. The pitfall is that you might be tempted to use them without trying to understand what they do. For this reason, in some examples I'll build the code manually instead of using the corresponding Wizard.

The Object Repository

Delphi has several menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File menu, and also in other pull-down menus. What happens if you simply select File | New | Other (from Delphi 6, or File | New in earlier versions)? Delphi opens the Object Repository (also called the New dialog box). The Object Repository is used to create new elements of any kind: forms, applications, data modules, libraries, thread objects, components, automation objects, and more. The Object Repository dialog box has a number of pages:

- The *New* and *ActiveX* pages allow you to create many different types of new items. At times when you create a new item, Delphi asks you the name of a new class and few other things, in a sort of *mini-Wizard*.
- The "current project" page (actually you'll see the name of the project) allows you to inherit from a form or data module included in your current project. In this page you can create new

forms or data modules that inherit from those of the current project. The content of this page depends exclusively on the units included in the current project. Simply create a couple of forms and then return to this page, and you'll see that its contents have already changed.

- The *Forms*, *Dialogs*, and *Data Modules* pages allow you to create a new element of these kinds starting from an existing one or using a Wizard.
- The *Projects* page allows you to copy the files from an existing project stored in the Repository, or use the Application Wizard.
- The *Multitier, Business, WebSnap,* and *WebServices* pages provide a starting point for using some of the advanced features found only in the Enterprise edition (and mostly only starting with Delphi 6)

Use the radio buttons at the bottom of the Object Repository dialog box to indicate that you want to copy an existing item, inherit from it, or use it directly without making a copy.

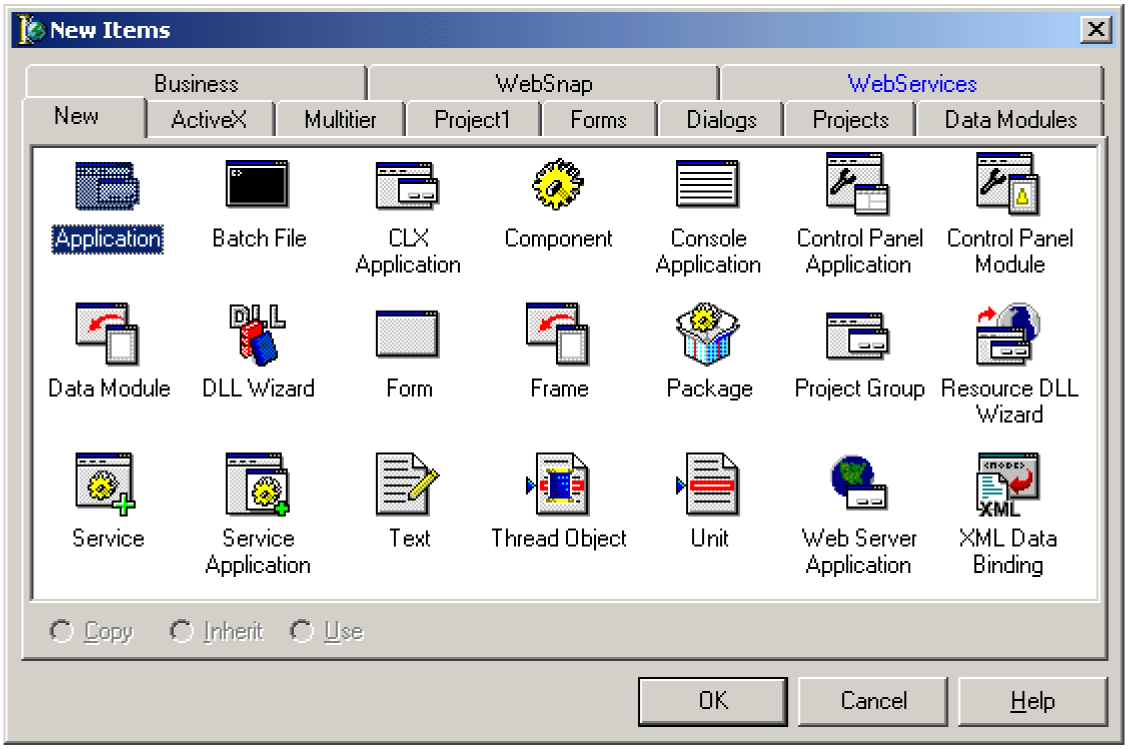
> The concept of inheritance in object-oriented programming is discussed in Mastering Delphi. In short, it is a way to add new capabilities to an existing form (or class in general) without making a full copy. This way, if you make a change in the original form (or class), the inherited form (or class) will be affected, too.

When you select a Wizard instead of a template, the only available radio button is *Copy*, meaning you'll end up with a new copy, the generated code. Keep in mind that I am discussing the pages of the Object Repository as they appear in Delphi 6 Enterprise, but different editions and versions of Delphi have fewer or different pages and items; and you can further customize the Repository, as we will see later on in this chapter.

> The Object Repository has a local menu that allows you to sort items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, details). This last view is the only one that gives you the description, the author, and the date of the tool. This information is particularly important when looking at Wizards, projects, or forms you've added to the Repository.

The New Page

The New page of the Object Repository (shown below) allows you to create several new items of the more commonly used kinds and is often an alternative to a direct menu command. Here is a list of the elements you can create from this page:



- *Application* creates a new blank project (the same as the command File | New | Application).
- *Batch File* opens a batch file you can include in a project group for automatic processing (to fully automate a complex build and deploy process).
- *CLX Application* creates a new blank project based on the CLX library (and on Qt), so that it will be fully portable to Kylix on the Linux platform (the same as the command File | New | CLX Application).
- *Component* creates a new Delphi component after you've completed the information requested by the simple Component Wizard. The same wizard can be activated with the Component | New Component menu command.
- *Console Application*
- *Control Panel Application*
- *Control Panel Module*
- *Data Module* creates a new blank data module (the same as the command File | New Data Module).
- *DLL Wizard* creates a simple DLL skeleton.
- *Form* creates a new blank form (the same as the command File | New Form).
- *Frame*
- *Package* creates a new Delphi component package. (You can also create a new package when creating a component.)

- *Project Group*
- *Resource DLL Wizard*
- *Service*
- *Service Application*
- *Text* opens a new ASCII text file in the Delphi editor.
- *Thread Object* creates a new thread object after asking you to fill in the New Thread Object dialog box. Multithreading in Windows is introduced in Chapter 25.
- *Unit* creates a new blank unit, a Pascal source file not connected with a form.
- *Web Server Application* allows you to create an ISAPI/NSAPI add-in DLL, a CGI stand-alone executable, a Win-CGI stand-alone executable, an Apache module, or a server application based on Delphi's Web Debugger. In each case Delphi creates a simple project based on a Web module (a special type of data module) instead of a form.
- *XML Data Binding*

*[***Many of the descriptions for newer entries are still missing]*

As some of the features made available by the Object Repository are quite advanced, it doesn't make

The Forms Page

This page lists predefined forms. Here is a short list of the predefined forms available in Delphi:

- *About box* is a simple About box.
- *Dual list box* is a form with two different list boxes, allowing a user to select a number of elements from one list and move them to the other list by pressing a button. Along with the components, this form contains a good amount of nontrivial Pascal code.
- *QuickReport Labels* creates a report form based on the QuickReport component.
- *QuickReport List* creates another form based on QuickReport, with a different layout.
- *QuickReport Master/Detail* is a third predefined report form with a more complex structure.
- *Tabbed pages* is a form based on the Windows PageControl.

Wizards can only be executed. The other forms, by contrast, can be used in multiple ways. You can add them into a project, inherit from them (in this case the original form will be automatically added to your project), or use them directly. When you use a form or inherit from one, take care not to make any change on the original forms in the Repository.

The Dialogs Page

This page is similar to the previous one, but includes forms that are typically used as dialog boxes. Here is the list of its items:

- *Dialog with help* is available in two versions. One has the buttons on the right side of the form (*Vertical*), and the other has them in the lower portion (*Horizontal*), as you can see from the corresponding icons.
- *Dialog Wizard* is a simple Wizard capable of generating different kinds of dialog boxes with one or more pages, as we will see later in this chapter.
- *Password dialog* is a dialog box with a simple edit box for entering a password.
- *Reconcile Error Dialog* is used in MIDAS or DataSnap applications.
- *QuickReport Wizard* creates a Print Options dialog box for a report.
- *Standard dialog* is also available in two versions, again *Horizontal* and *Vertical*, with buttons in different positions.

The Data Modules Page

You already know what a project and a form are, but what is a data module? It is a sort of form that never appears on screen at run-time and can be used to hold nonvisual components. It is mostly used to implement code related to database access. This page has only a data module at start-up, *Customer Data*. If you have several forms or applications accessing the same data tables and database queries, you can easily define new data modules and add them to the repository.

The Projects Page

The last page we will look at in this introduction contains project schemes you can use as the starting point for your own application. These projects often include one or more forms. Here is the list of them:

- *Application Wizard* is another simple Wizard that allows you some limited choices for the file structure and other elements of an application.
- *MDI Application* defines the main elements of a Multiple Document Interface (MDI) program. It defines a main form for the MDI frame window, with a menu, a status bar, and a toolbar. It also defines a second form that can be used at run-time to create a number of child windows.
- *SDI Application* defines a main form with the standard attributes of a modern user interface, including a toolbar and a status bar, and also a typical About box.
- *Win2000 Logo Application* defines a sample application with most of the elements required by Microsoft for an application to get the Windows 2000 compatibility logo. This command basically creates an SDI application with a RichEdit component in it, and adds the code needed to make the application mail-enabled.

- *Win95 Logo Application* defines a sample application with most of the elements required by Microsoft for an application to get the Windows 95/98 compatibility logo. The application is similar to the one above.

When you select one of these projects, Delphi asks you to enter the name of an existing or a new directory. If you indicate a new directory, Delphi will automatically create it.

For example, we can create an SDI project based on the corresponding template. Then we can customize it, giving it a proper title, removing or adding new components, and writing some code. Some interesting components are already there, however, and there is even some ready-to-use code to make those components work properly. The menu and toolbar of the application, can be used to open some dialog boxes. File Open and File Save dialog boxes are wrapped up in components that are added to the form by the template; the About box is defined by the template as a second form.

In the simple SdiTemp example, I've decided to make just a few limited changes: I've entered a new title for the main form and some text for the labels for the About box (the property to use is Caption in both cases). The result is the application shown in *[***missing figure]* and available in the source code.

Delphi Wizards

Besides copying or using existing code, Delphi allows you to create a new form, application, or other code files, using a Wizard. Wizards (or Experts) allow you to enter a number of options and produce the code corresponding to your choices.

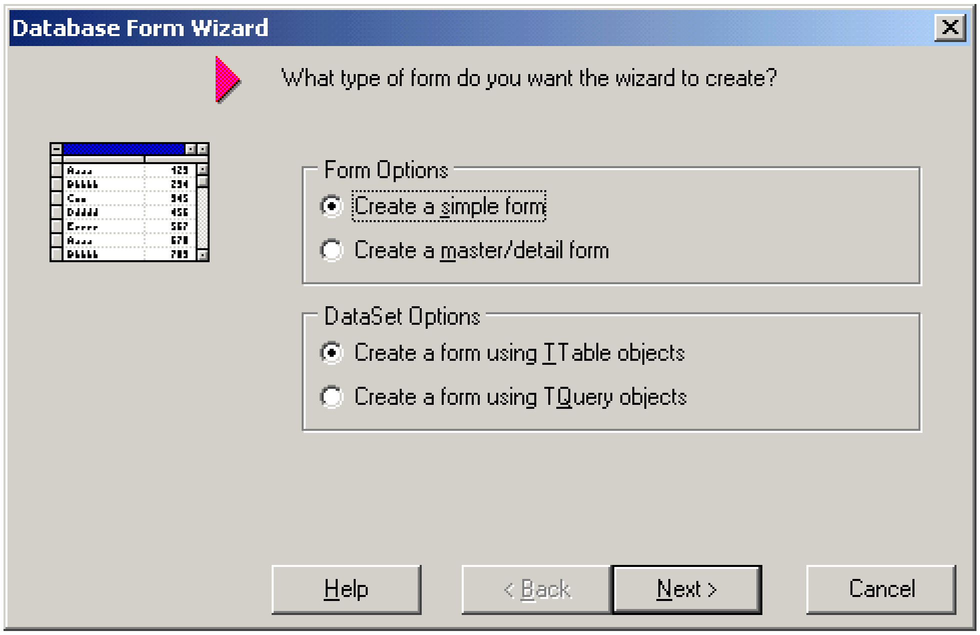
One of the most important predefined wizards is the Database Form Wizard, which you can activate using the Database | Form Wizard menu item or the icon in the Forms page of the Object Repository. There are also some other simple Wizards in Delphi, such as the Application Wizard and the Dialog Wizard. I've listed various other Wizards in the description of the pages of the Object Repository, in the last section.

You can also buy add-on Wizards from third-party tool providers, download one from my web site, or even write your own Wizard. Add-on Wizards often show up in the Help menu, but it is possible to add new menu items in other Delphi pull-down menus or install Wizards in various pages of the Object Repository.

The Database Form Wizard

In this section, I'll show you a quick example of the use of the Database Form Wizard, but I won't describe the application we build in detail. In this example, we'll build a database program using some of the data already available in Delphi. Note that you have to create a project first, and then start the Database Form Wizard. So you usually end up with two forms, unless you remove the original form from the project. Fortunately, one of the Wizard's options, displayed at the end, lets you select the new form generated by the Wizard as the main form.

1. As soon as you start the Database Form Wizard, you will be presented with a number of choices, which depend on the options you choose at each step. The first page lets you choose between a simple or a master detail form, and between the use of tables or queries. Leave the selections as they are by default, and move on by clicking on the Next button.



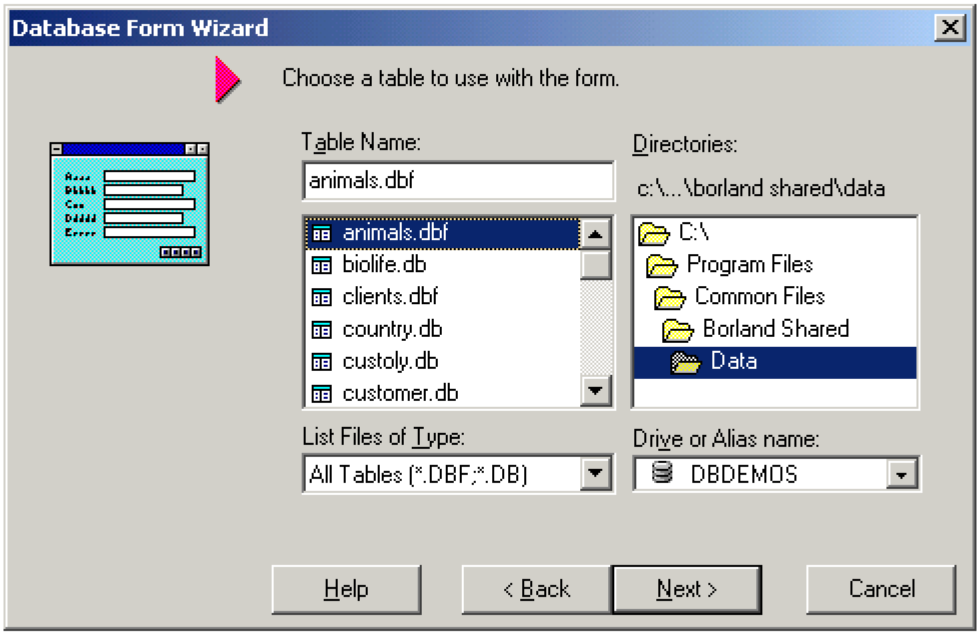
Database Form Wizard

What type of form do you want the wizard to create?

Form Options
● Create a simple form
○ Create a master/detail form

DataSet Options
● Create a form using TTable objects
○ Create a form using TQuery objects

- **2.** In the next page you can choose an existing database table to work on. In the Drive or Alias Name combo box, there should be a DBDEMOS alias. After you select this option, a number of Delphi demo database tables appear in the list. Choose the first, `animals.dbf`.



Database Form Wizard

Choose a table to use with the form.

Table Name: animals.dbf

animals.dbf
biolife.db
clients.dbf
country.db
custoly.db
customer.db

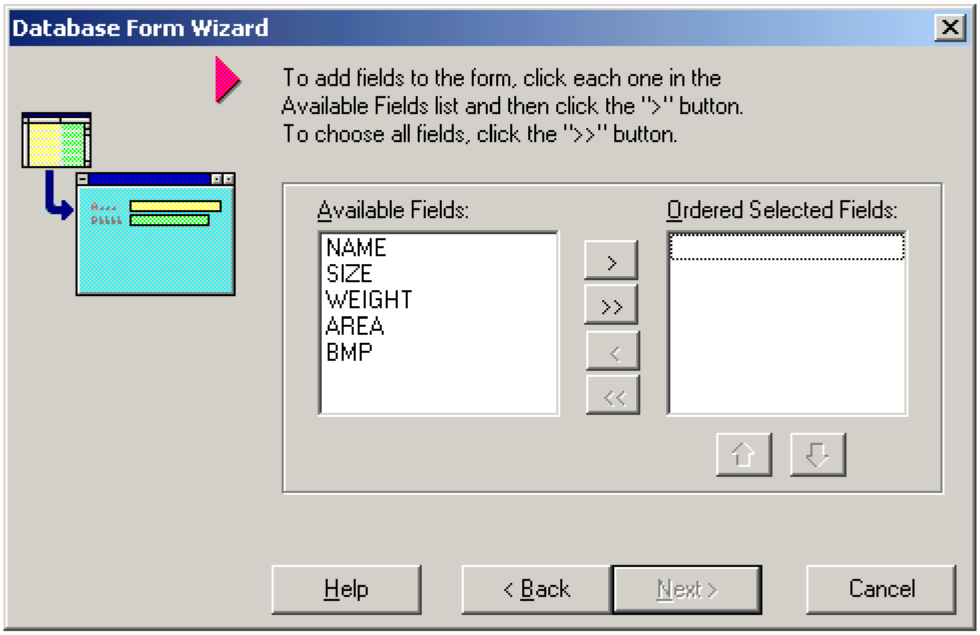
Directories: c:\...\borland shared\data

C:\
Program Files
Common Files
Borland Shared
Data

List Files of Type: All Tables (*.DBF;*.DB)

Drive or Alias name: DBDEMOS

- **3.** In the third page, you can choose the fields of the selected database table that you want to consider. To build this first example, choose all of the fields by clicking on the >> button.



Database Form Wizard

To add fields to the form, click each one in the Available Fields list and then click the ">" button. To choose all fields, click the ">>" button.

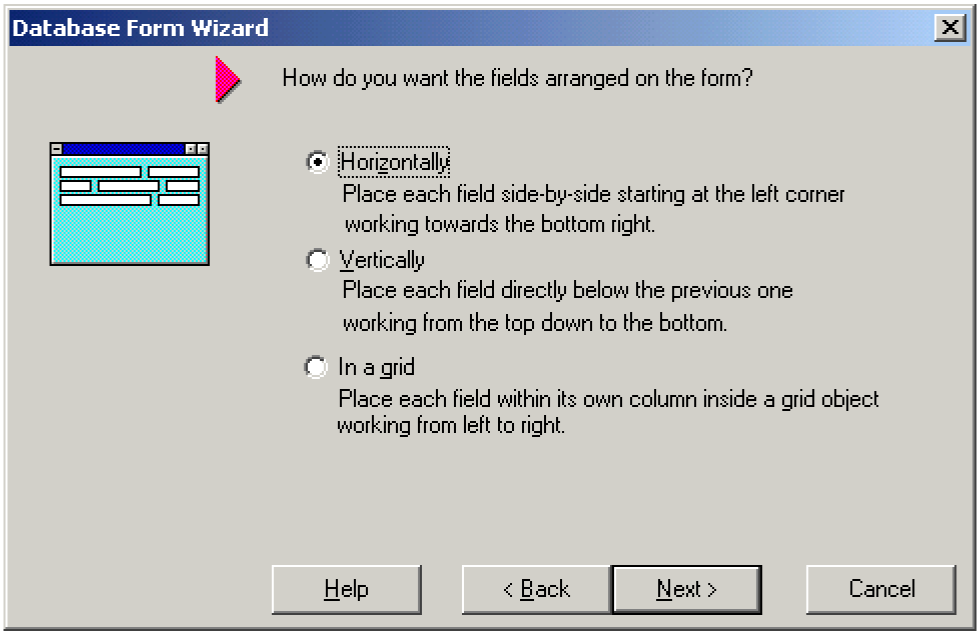
Available Fields:

- NAME
- SIZE
- WEIGHT
- AREA
- BMP

Ordered Selected Fields:

-

4. On the next page, you can choose from various layouts. If you choose Vertical, the next page will ask you the position of the labels. The default option, Horizontal, might do.



Database Form Wizard

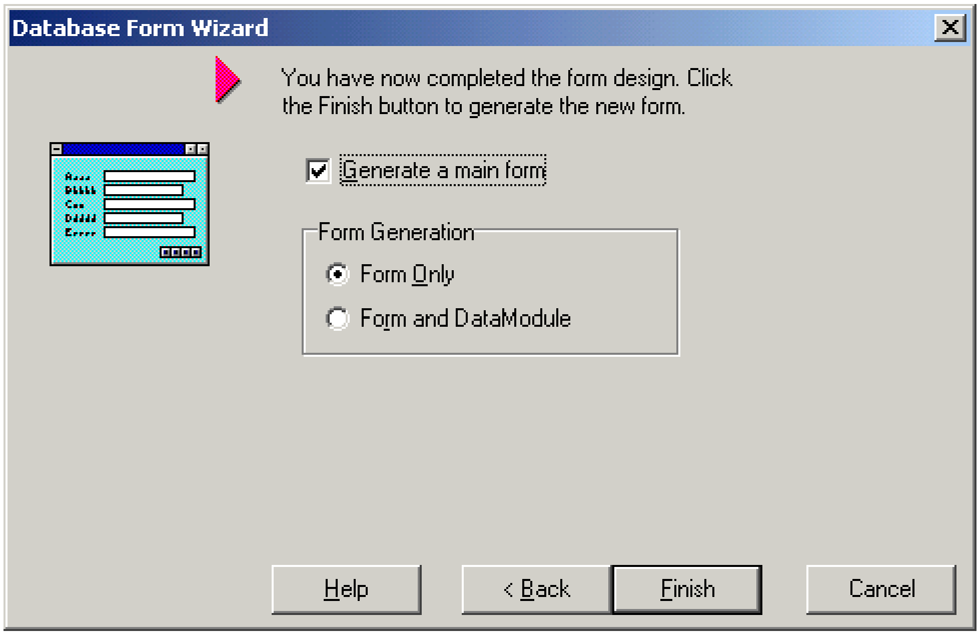
How do you want the fields arranged on the form?

● Horizontally
Place each field side-by-side starting at the left corner working towards the bottom right.

○ Vertically
Place each field directly below the previous one working from the top down to the bottom.

○ In a grid
Place each field within its own column inside a grid object working from left to right.

- **5.** The next page is the last. Leave the Generate a Main Form check box and the Form Only radio button selected, and click on the Finish button.



Database Form Wizard

You have now completed the form design. Click the Finish button to generate the new form.

☑ Generate a main form

Form Generation

● Form Only

○ Form and DataModule

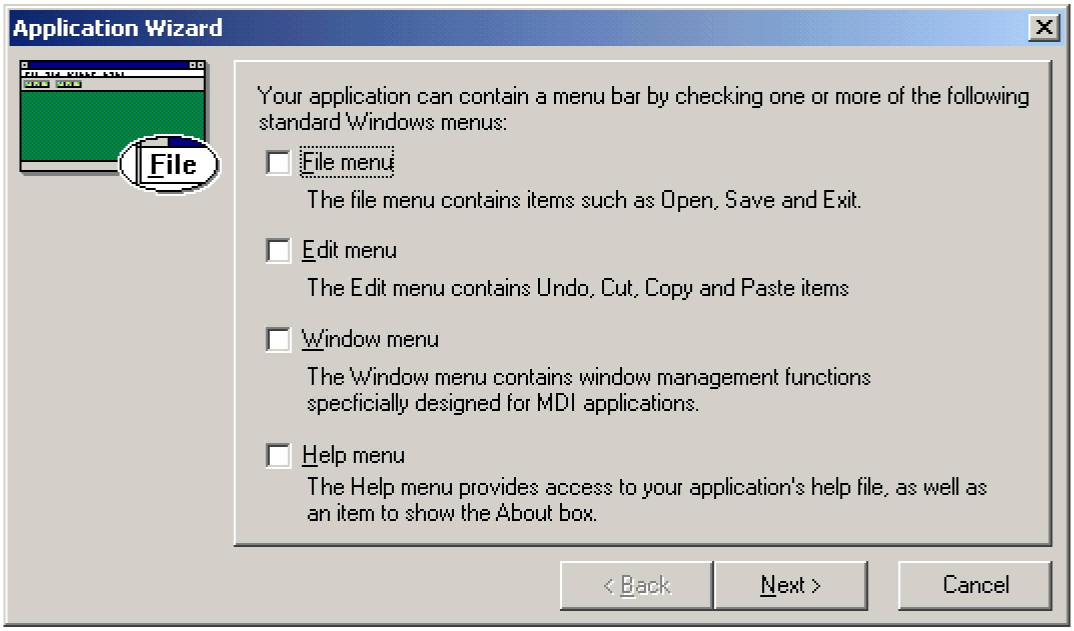
You can immediately compile and run the application. The result is a working database application, which allows you to navigate among many records using the buttons. This specific application (the DataExp example) even has a graphical field, displaying a bitmap with the current animal.

The output of the generated form is usually far from adequate. In this case, the image area is too small; at other times the positioning of the controls may not be satisfactory. Of course, you can easily move and resize the various components placed on the form at design-time.

To make this task easier, you can select the Table component (the one in the upper-left corner of the form) and toggle its Active property to True. Then the data of the table's first record will be displayed at design-time. This is helpful because it allows you to see an example of the length of the field's text and the size of the image. Note that the Database Form Wizard generates almost no Pascal code, besides a line used to open the table when the program starts. The capabilities of the resulting programs stem from the power of the database-related components available in Delphi.

The Application Wizard

Another interesting (although less powerful) tool is the Application Wizard. You can activate it from the Projects page of the Object Repository. The Application Wizard allows you to create the skeleton of a number of different kinds of applications, depending on the options you select.



Your application can contain a menu bar by checking one or more of the following standard Windows menus:

□ File menu

The file menu contains items such as Open, Save and Exit.

□ Edit menu

The Edit menu contains Undo, Cut, Copy and Paste items

□ Window menu

The Window menu contains window management functions specficially designed for MDI applications.

□ Help menu

The Help menu provides access to your application's help file, as well as an item to show the About box.

The first page of this Wizard allows you to add some standard pull-down menus to the program: File, Edit, Window, and Help. If you select the File menu, the second page will ask you to enter the file extensions the program should consider. You should enter both a description of the file, such as *Text file (*.txt)*, and the extension, *txt*. (You can input several extensions, each with its own description.) These values will be used by the default File Open and File Save dialog boxes that the Application Wizard will add to the program if you select the file support option.

Then, if you have selected any of the pull-down menus, the Application Wizard displays a nice visual tool you can use to build a toolbar. Unfortunately, this tool is not available as a separate editor inside Delphi. You simply select one of the pull-down menus, and a number of standard buttons corresponding to the typical menu items of this pull-down menu appear (but only if the menu has been selected on the first page of the Application Wizard).

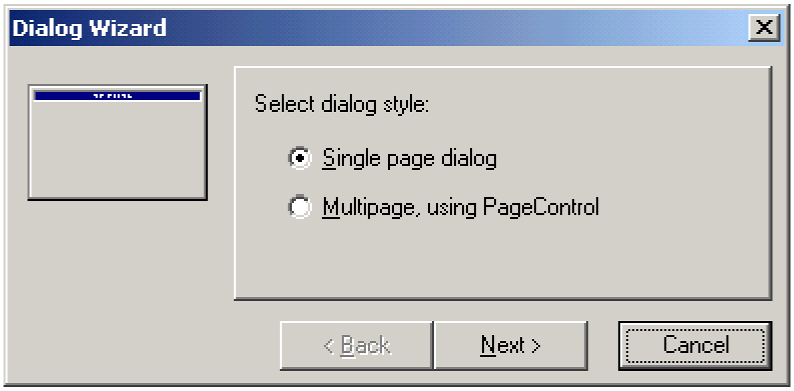
To add a new button, select one of them in the graphical list box on the right and press the Insert button. The new toolbar button will be added at the position of the small triangular cursor. You can move this cursor by clicking on one of the elements already added to the toolbar. This cursor is also used to indicate a button you want to remove from the toolbar.

When the toolbar is finished, you can move to the last page. Here you can set several other options, such as choosing MDI support, adding a status bar, and enabling hints. You can also give a name to the new application and specify a directory for the source files. The name of the application can be long, but it cannot contain white spaces (it should be a valid Pascal identifier), and the directory for the application should be an existing directory. To place the project files in a new directory, choose the Browse button, enter the new path, and the dialog box will prompt you to create the new directory.

Although it is somewhat bare and it has room for improvement, the Delphi Application Wizard is much more useful than the predefined application templates for building the first version of an application. One of its biggest advantages is that you can define your own toolbar. Another advantage is that the Application Wizard generates more code (and more comments) than the corresponding templates do. The disadvantage of this Wizard is that it generates an application with a single form. Its MDI support is limited, because no child form is defined, and the generated application has no About box.

The Dialog Wizard

Delphi's Dialog Wizard is a simple Wizard provided mostly as a demo, with its own source code. From the code of this Wizard, in theory you should be able to learn how to build other Wizards of your own. However, you can still use the Dialog Wizard as a tool to build two kinds of dialog boxes: simple dialog boxes and multiple-page dialog boxes based on the Windows PageControl component.



Select dialog style:

● Single page dialog

○ Multipage, using PageControl

If you choose the simple dialog box, the Wizard will jump to the third page, where you can choose the button layout. If you choose the multiple-page dialog box, an intermediate page will appear to let you input the text of the various tabs. This Wizard is an alternative to the corresponding form templates of the Object Repository. Its advantage is that it allows you to input the names of the PageControl tabs directly.

Customizing the Object Repository

Since writing a new wizard is far from simple, the typical way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the *New* and "current project" pages).

Adding New Application Templates

Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of two or more similar programs, you can save the current status to a template, ready to use later on.

> Although Borland calls everything you can put in the Object Repository an *object*, from an object-oriented perspective this is far from true. For this reason I call the schemes you can save to disk for later use *templates*. Application templates, in particular, do not relate to objects or classes in any way, but are copied to the directory of your new project. *Object Repository* sounds much better than *Browse Gallery* (the name used in Delphi 1), but besides the capability to activate form inheritance, there is not much object-oriented in this tool.

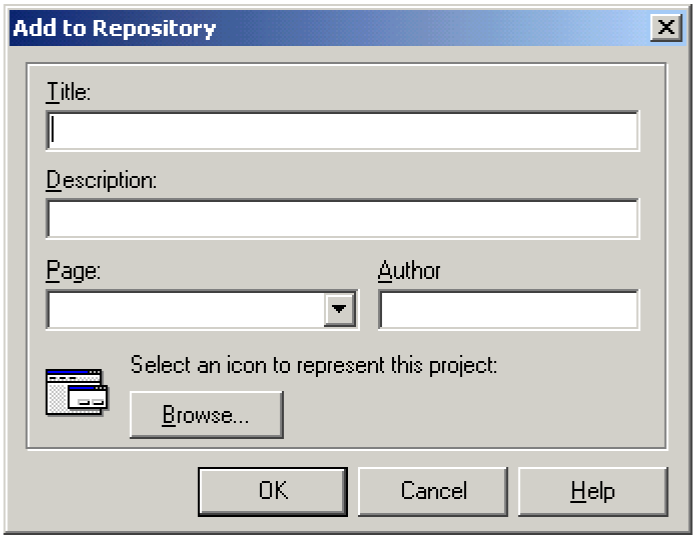
You can add a project to the Repository by using the Project | Add to Repository command, or by using the corresponding item of the local menu of the Project Manager window. As a very simple example, just to demonstrate the process, the following steps describe how you can save the slightly modified version of the default SDI template (shown earlier) as a template:

1. Open the modified SdiTemp example (or any other project you are working on).

2. Select the Project | Add to Repository menu command (or the Add Project to Repository command in the local menu of the Project Manager window).

3. In the Add to Repository dialog box, enter a title, a description for the new project template, and the name of the author. You can also choose an icon to indicate the new template or accept the default image. Finally, choose the page of the Repository where you wish to add the project.

4.: Click on OK, and the new template is added to the Delphi Object Repository.



Now, each time you open the Object Repository, it will include your custom template. If you later discover that the template is not useful any more, you can remove it. You can also use a project template to make a copy of an existing project so that you can continue to work on it after saving the original version.

However, there is a simpler way to accomplish this: copy the source files to a new directory and open the new project. If you do copy the source files, do *not* copy the DSK file, which indicates the position of the Windows on the screen. The DSK file holds a list of files open in the editor, using an absolute path. This means that as soon as you open the new project and start working on it, you may well end up editing the source code files of the original project and compiling the files of the new version (the project manager stores relative paths).

This will certainly surprise you when the changes you make in code or in forms seem to have no effect. Simply deleting the DSK file, or not copying it in the first place, avoids this problem.

The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or wizards, this is not what you want. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple:

1: Create a new project as usual.

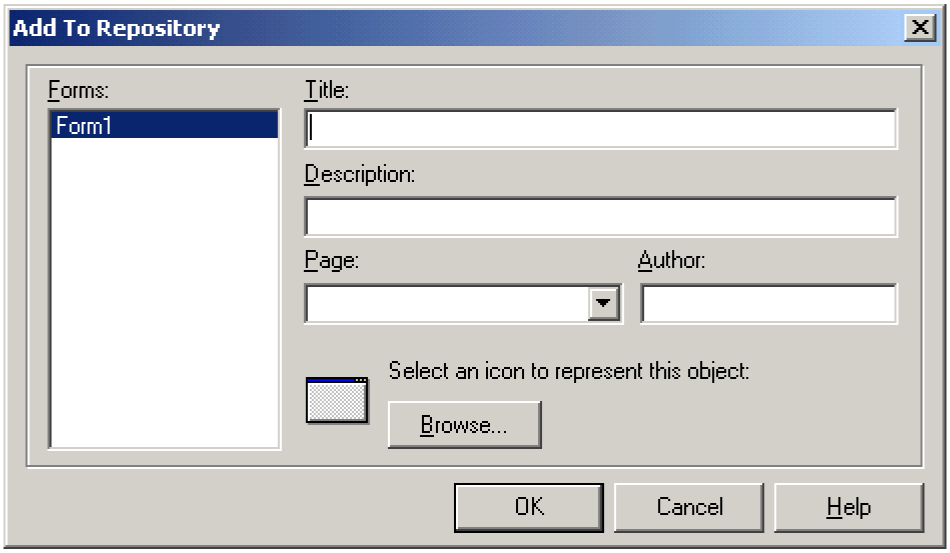
2: Remove its only form from the project.

3: Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages. You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you need to use the File | Save Project As command to give a new name to the project, since saving the project automatically uses the default name in the template.

Adding New Form Templates to the Object Repository

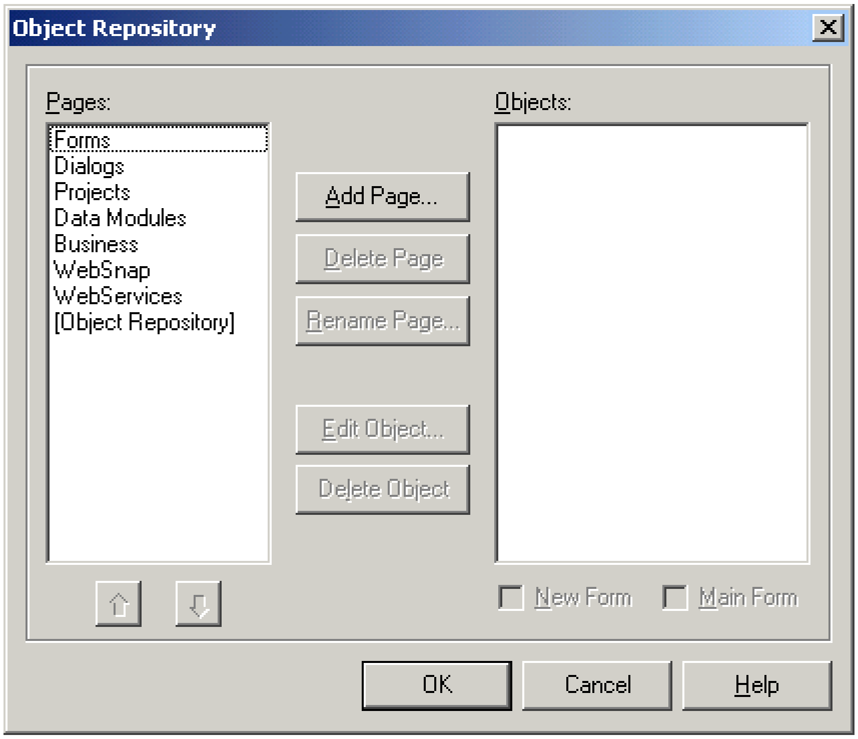
Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form you want to add, right-click on it, and select Add to Repository from the local menu. In the dialog box that appears (see below), you can choose which form of the current project should be added to the Repository, and set the title, description, author, page, and icon, as usual. Once you have set these elements and clicked on OK, the form is added to the proper page of the Object Repository.



This approach is suggested if you have a complex form to make available to other applications and other programmers. They will be able to use your form as is, make a copy of it, and inherit from it. For this reason, adding forms to the Repository is far more flexible than adding projects, which can only be copied as the starting point of a new application.

The Object Repository Options

To further customize the Repository, you can use the Tools | Repository command to open the Object Repository dialog box. This dialog box is quite easy to use; on the left is the list of current Repository pages and on the right the list of items in each page, including both templates and Wizards. You can use this dialog to move repository items to different pages, to add new elements, or to delete existing ones.



You can use the three page-related buttons and the two buttons with arrows below the list of pages to arrange the structure of the Object Repository, adding new pages, renaming or deleting them, and changing their order. All these operations affect some of the tabs of the Object Repository itself (other tabs are fixed).

An important element of the Object Repository setup is the use of defaults:

- Use the *New Form* check box below the list of objects to designate the current form or Wizard as the default, to be used when a new form is created (File | New | Form). Only one object in the Object Repository can be used as a default; it is marked with a special symbol placed over its icon.

- The *Main Form* check box, instead, is used to indicate the form or Wizard used to create the main form of a new application (File | New | Application) when no special New Project is selected (see next bullet). The current Main form is indicated by a second special symbol.
- The New Project check box, available when you select a project object, can be used to mark the default project that Delphi uses when you issue the File | New | Application command. Also, the New Project is indicated by its own special symbol.

If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the OBJREPOS subdirectory of the Delphi main directory. At the same time, if you directly use a form or any other object without copying it, then you end up having some files of your project in this directory. It is important to realize how the repository works, because if you want to modify a project or an object saved in the repository, the best approach is to operate on the original files, without copying data back and forth to the Repository.

Installing new DLL Wizards

Technically, new Wizards come in two different forms. Wizards may be part of components or packages, and in this case are installed the same way you install a component or a package. Other Wizards are distributed as stand-alone DLLs. In this case you should add the name of the DLL in the Windows Registry under the key:

Software\Borland\Delphi x.0\Experts.

Simply add a new string key under this, choose a name you like (it doesn't really matter) and use as text the path and filename of the Wizard DLL. You can look at the entries already present under the Experts key to see how the path should be entered.

What's Next

In this short chapter, we have seen how you can start the development of an application by using Delphi templates and Wizards. You can use one of the predefined application templates or form objects, start the Database Form Wizard, or use the other Wizards for a fast start with applications, forms, and other objects.

However, I'll rarely use these templates and Wizards in the rest of the book. With the exception of the Database Form Wizard, these tools let you build only very simple applications, which you can often put together yourself in seconds when you are an experienced Delphi programmer. For beginners, Wizards and templates provide a quick start in code development. But you are not going to *remain* a beginner, are you?

So with the next chapter we'll start focusing on the use of the actual components and controls available in Delphi, to build actual applications, even if simple ones. Differently form other books of mine, which focus more on the foundations, this text proceeds mainly though practical examples.

Notice that the only reason there is no introduction to the use of the language and the core structure of the VCL, is that because you can find these topics covered in Essential Pascal (for the core language) and Mastering Delphi (for the OOP features and the VCL architecture).

Chapter 4: A Tour of the Basic Components

- Clicking a button or another component
- Adding colored text to a form
- Dragging from one component to another
- Accepting input from the user
- Creating a simple editor
- Making a choice with radio buttons and list boxes
- Allowing multiple selections
- Choosing a value in a range

Now that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the Visual Component Library, we are ready to delve into the central part of the book: the use of components. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

The system comes with a number of ready-to-use components. I will not describe every component in detail, examining each of its properties and methods. If you need this information, you can find it easily in the Help system. The aim of Part II of this book is to show you how to use some of the features offered by the Delphi predefined components to build applications. In fact, this chapter and those following will be based heavily on sample programs. These examples tend to be quite simple — although I've tried to make them meaningful — in order to focus on only a couple of features at a time.

I'll start by focusing on a number of basic components, such as buttons, labels, list boxes, edit fields, and other related controls. Some of the components discussed in this chapter are present in the Standard page of the Delphi Components palette; others are in different pages. I'm not going to describe all the components of the Standard page, either. My approach will be to discuss, in each chapter, logically related components, ignoring the order suggested by the pages of the Components palette.

Windows Own Components

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window with a specific behavior, some properties, and some methods (although traditional C language code used to access the predefined components in Windows by sending and receiving messages). These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components.

Actually Microsoft's third step is its ActiveX controls (an extension of the older OCX technology), the natural successor of VBX controls. In Delphi you can use both ActiveX and native components, but if you look at the technology, Delphi components are really ahead of the ActiveX controls. It is enough to say that Delphi components use OOP to its full extent, while ActiveX controls do not fully implement the concept of inheritance. With the recent introduction of the .NET framework, Microsoft has finally followed Borland and Sun (with Java) with the adoption of a technology that treats components as classes and vice verse.

Windows 3 had six kinds of predefined controls, generally used inside dialog boxes. They were buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Windows 95 added a number of new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. These controls were already used by programmers, who had to re-implement them each time. Windows 95 developers could for the first time use the standard common controls provided by the system, and starting with Delphi 3 developers had the advantage of having corresponding easy-to-use components.

The standard system controls are the basic components of each Windows application, regardless of the programming language used to write it, and are very well known by every Windows user. Delphi literally wraps these Windows predefined controls in some of its basic components — including those discussed in this chapter.

Notice that CLX provides components similar to Windows common controls that are portable to Kylix. This is important as Linux, of course, hasn't got Microsoft's common controls!

Clicking a Button

In the first chapter of this book, we built small applications based on a button. Clicking on that button caused a "*Hello*" message to appear on the screen. The only other operation that program performed was moving the button so that it always appeared in the middle of the form. Since then we've seen other examples that used buttons as a way to perform a given action (using their `OnClick` event).

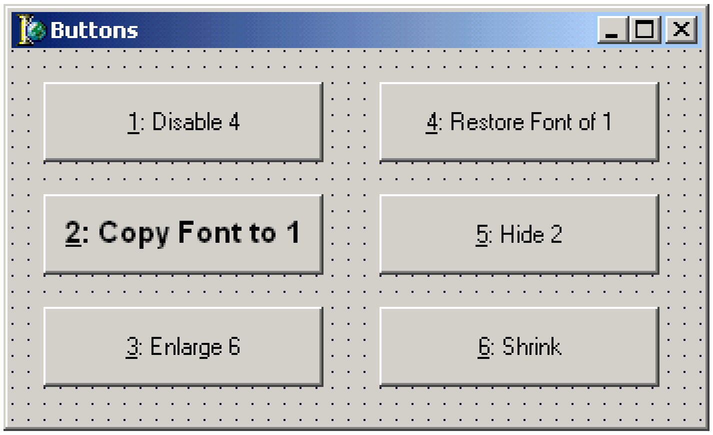
Now we are going to build another form with several buttons and change some of their properties at run-time; in this example, clicking a button will usually change a property of another button. To build the Buttons program, I suggest you follow the instructions closely at first and then make any changes you want. Of course, you can read the description in the book and then work on the source files, if you prefer.

The Buttons Example

First, open a new project and give it a name by saving it to disk. I've given the name `ButtonF.pas` to the unit describing the form and the name `Buttons.dpr` to the project. Now you can create a number of buttons, let's say six.

> Instead of selecting the component, dragging it to the form, and then selecting it again to repeat the operation, you can take a shortcut. Simply select the component by clicking on the Components palette while holding down Shift. The component will remain selected, as indicated by a little border around it. Now you can create a number of instances of that component.

Even if you use the grid behind the form, you might need to use the Edit | Align command to arrange the buttons properly. Remember that to select all six buttons at a time, you can either drag a rectangle around them or select them in turn, holding down the Shift key. In this case, it's probably better to select a column of three buttons at a time and arrange them.



Now that we have a form with six buttons, we can start to set their properties. First of all, we can give the form a name (`ButtonsForm`) and a caption (*Buttons*). The next step is to set the text, or `Caption` property, of each button. Usually, a button's `Caption` describes the action performed when a user clicks on it. We want to follow this rule, adding the number of the button at the beginning of each caption. So if the first button disables button number four (which is the one on the same row), we can name it *1: Disable 4*. Following the same rule, we can create captions for the other buttons.

For a summary of the properties of the components of this form, you can refer to its textual description:

```
object ButtonsForm: TButtonsForm
  Caption = 'Buttons'
  object Button1: TButton
    Caption = '&1: Disable 4'
    OnClick = Button1Click
  end
  object Button2: TButton
    Caption = '&2: Copy Font to 1'
    Font.Color = clBlack
    Font.Height = -15
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ParentFont = False
    OnClick = Button2Click
  end
  object Button3: TButton
    Caption = '&3: Enlarge 6'
    OnClick = Button3Click
  end
  object Button4: TButton
    Caption = '&4: Restore Font of 1'
    OnClick = Button4Click
  end
  object Button5: TButton
    Caption = '&5: Hide 2'
    OnClick = Button5Click
  end
  object Button6: TButton
    Caption = '&6: Shrink'
    OnClick = Button6Click
  end
end
```

Notice that every button has an underlined shortcut key, in this case the number of the button. Simply by placing an ampersand (&) character in front of each caption, as in *'&1: Disable 4'*, we can create buttons that can be used with the keyboard. Just press a number below 7, and one of the buttons will be selected, although you won't see it pressed and released.

The final step, of course, is to write the code to provide the desired behavior. We want to handle the OnClick event of each button. The easiest code is that of Button2 and Button4. When you press Button2, the program copies the font of this button (which is different from the standard font of the other buttons) to Button1, and then disables itself:

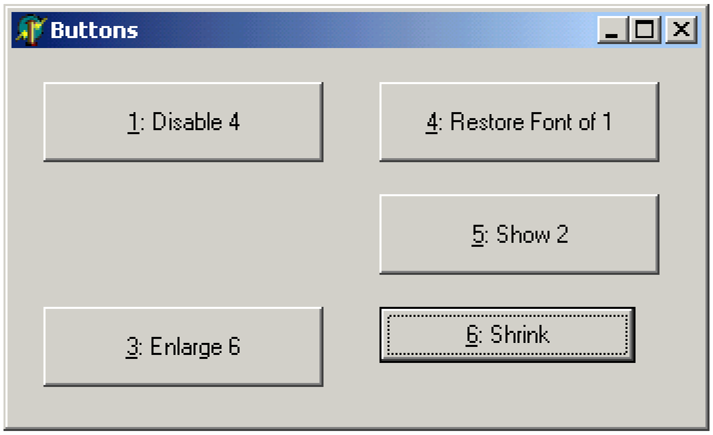
```
procedure TButtonsForm.Button2Click(Sender: TObject);
begin
  Button1.Font := Button2.Font;
  Button2.Enabled := False;
end;
```

Pressing Button4 restores the original font of the button. Instead of copying the font directly, we can restore the font of the form, using the ParentFont property of the button. The event also enables Button2, so that it can be used again to change the font of Button1:

```
procedure TButtonsForm.Button4Click(Sender: TObject);
begin
  Button1.ParentFont := True;
  Button2.Enabled := True;
end;
```

To implement the Disable and Hide operations of Button1 and Button5, we might use a Boolean variable to store the current status. As an alternative, we can decide which operation to perform while checking the current status of the button — the status of the Enabled property. The two methods use two different approaches, as you can see in the following code:

```
procedure TButtonsForm.Button1Click(Sender: TObject);
begin
  if not Button4.Enabled then
  begin
    Button4.Enabled := True;
    Button1.Caption := '&1: Disable 4';
  end
  else
  begin
    Button4.Enabled := False;
    Button1.Caption := '&1: Enable 4';
  end;
end;

procedure TButtonsForm.Button5Click(Sender: TObject);
begin
  Button2.Visible := not Button2.Visible;
  if Button2.Visible then
    Button5.Caption := '&5: Hide 2'
  else
    Button5.Caption := '&5: Show 2';
end;
```



You can see the results of this code in the figure above. The last two buttons have *unconstrained* code. This means that you can shrink Button6 so much that it will eventually disappear completely:

```
procedure TButtonsForm.Button3Click(Sender: TObject);
begin
  Button6.Height := Button6.Height + 3;
  Button6.Width := Button6.Width + 3;
end;

procedure TButtonsForm.Button6Click(Sender: TObject);
begin
  Button6.Height := Button6.Height - 3;
  Button6.Width := Button6.Width - 3;
end;
```

It would have been quite easy, in any case, to check the current size of the button and prevent its reduction or enlargement by more than a certain value.

Clicking the Mouse Button

Up to now we have based the examples on the OnClick event of buttons. Almost every component has a similar click event. But what exactly is a click? And how is it related to other events, such as OnMouseDown and OnMouseUp?

First, consider the click. At first sight you might think that to generate a click, a user has to press and release the left mouse button on the control. This is certainly true, but the situation is more complex. When the user clicks the left mouse button on a button component, the component is graphically pressed, too. However, if the user moves the cursor (holding down the left mouse button) outside the button surface, this button will be released. If the user now releases the left mouse button outside the button area, no effect — no click — takes place. On the other hand, if the user places the cursor back on the button, it will be *pressed* again, and when the mouse button is released, the click will occur. If this is not clear, experiment with a button; any button in any Windows application will do.

Now to the second question. In Windows, the behavior just described is typical of buttons, although Delphi has extended it to most components, as well as to forms. In any case, the system generates more basic events — one each time a mouse button is pressed, and another each time a button is released. In Delphi, these events are called OnMouseDown and OnMouseUp. Since the mouse has more than one button, these same events are generated when the user presses any mouse button.

You might want different actions to occur, depending on the mouse button. For this reason, these event handlers include a parameter indicating which button was pressed. These methods also include another parameter, indicating whether some special key (such as Shift or Ctrl) has been pressed and, finally, two more values indicating the *x* and *y* positions where the action took place. This is the method corresponding to this event for a form:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

Most of the time, we do not need such a detailed view, and handling the mouse-click event is probably more appropriate.

Adding Colored Text to a Form

Now that you have played with buttons for a while, it's time to move to a new component, labels. Labels are just text, or comments, written in a form. Usually, the user doesn't interact with a label at all — or at least not directly. It doesn't make much sense to click on a label (although in Delphi this is technically possible). Keep in mind, however, that not all of the text you see in a form corresponds to a label. A form (and any other component) can simply output text on its surface, for example using the TextOut method.

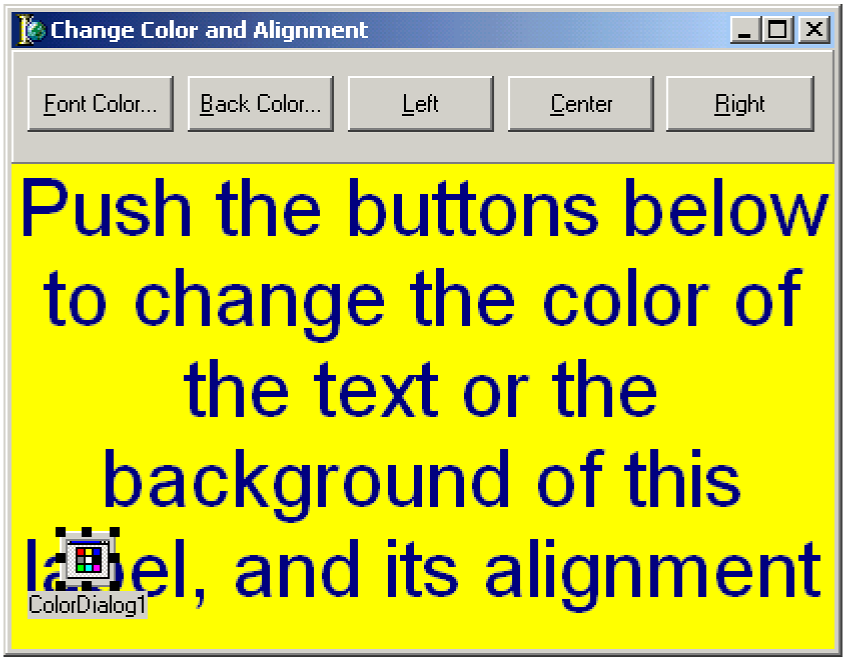
We use labels to provide descriptions of other components, particularly edit fields and list or combo boxes, because they have no title. If you open a dialog box in any Windows application, you'll probably see some text. These are *static controls* (in Windows terms) or *labels* (in Delphi terms).

Windows implements labels as windows of the static class. Delphi, instead, implements labels as non-windowed, graphical components. This is very important since it allows you to speed up form creation and save some Windows resources. However, Delphi also includes a new component that corresponds to the Windows label, the StaticText component. This component has similar properties and the same events, and Borland seems to have added it mainly for ActiveX support. We will use this component in the next example.

Besides using labels for descriptions, we can use instances of this component to improve and add some color to the user interface of our application. This is what we are going to do in the next example, LabelCo. The basic idea of this application is to test a couple of properties of the label component at run-time. Specifically, we want to alter the background color of the label, the color of its font, and the alignment of the text.

The LabelCo Example

The first thing to do is to place a big label in the form and enter some text. Write something long. I suggest you set the `WordWrap` property to `True`, to have several lines of text, and the `AutoSize` property to `False`, to allow the label to be resized freely. It might also be a good idea to select a large font, to choose a color for the font, and to select a color for the label itself.



Push the buttons below to change the color of the text or the background of this label, and its alignment

To change the font color, background color, and alignment properties of the label at run-time, we can use buttons. Instead of placing these buttons directly on the form, we can place a Panel component in the form, and then place the five buttons over (or actually inside) the panel. We need two to change the colors and three more to select the alignment — left, center, or right. Placing the buttons inside the panel, this last control will become the `Parent` component of the buttons: the coordinates of the buttons will be relative to the panel, so that moving the panel will move the buttons, hiding the Panel will hide the buttons, and so on.

> The difference between the `Parent` and the `Owner` properties of a component is very important. The first indicates visual containment (like a button being inside a panel) while the latter indicates construction and destruction ownership (like a button being owned and destroyed by a form).

Making the label and the panel child components of the form allows us to align them (something you cannot do with buttons). Simply set the `Align` property of the Panel to `alTop`, and the `Align` property of the Label to `alClient`, and the two components will take up the full client area of the form. What's interesting is that when we resize the form, the two components will adjust their size and position correspondingly. Notice, by the way, that the StaticText component has no `Align` property.

The last component we have to place on the form is a ColorDialog component (you can find it in the Dialogs page of the Components palette). This component invokes the standard Windows Color dialog box. The resulting form is detailed in the following listing:

```
object ColorTextForm: TColorTextForm
  Caption = 'Change Color and Alignment'
  object Label1: TLabel
    Align = alClient
    Alignment = taCenter
    Caption = 'Push the buttons...' // omitted
    Color = clYellow
    Font.Color = clNavy
    Font.Height = -40
    Font.Name = 'Arial'
    WordWrap = True
  end
  object Panel1: TPanel
    Align = alTop
    object BtnFontColor: TButton...
    object BtnBackColor: TButton...
    object BtnLeft: TButton...
    object BtnCenter: TButton...
    object BtnRight: TButton...
  end
  object ColorDialog1: TColorDialog...
end
```

> The two buttons used to change the color will display a dialog box, instead of performing an action directly. For this reason at the end of their `Caption` there is an ellipsis (...), which is the standard Windows convention for button and menu items to indicate the presence of a dialog box.

Now it's time to write some code. The click methods for the three alignment buttons are very simple. The program has to change the alignment of the label, as in:

```
procedure TColorTextForm.BtnLeftClick(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
end;
```

The other two methods should use the values `taCenter` and `taRightJustify` instead of `taLeftJustify`. You can find the names of these three choices in the `Alignment` property of the label, in the Object Inspector.

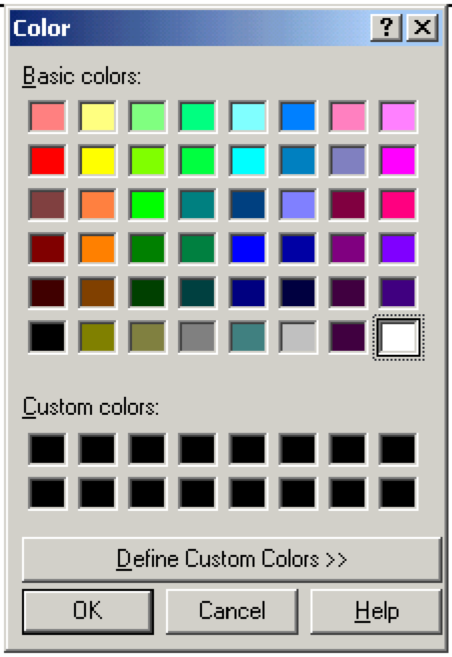
Writing code to change the color is a little more complex. In fact, we can provide a new value for the color, maybe choosing it from a list with a series of possible values. We might solve this problem, for example, by declaring an array of colors, entering a number of values, and then selecting a different element of the array each time. However, a more professional solution needs even less code: using the Windows standard dialog box to select a color.

The Standard Color Dialog Box

To use the standard Color dialog box, move to the Dialogs page of the Delphi Components palette, select the ColorDialog component, and place it anywhere on the form. The position has no effect, since at run-time this component is not visible inside the form. Now we can use the component, writing the following code:

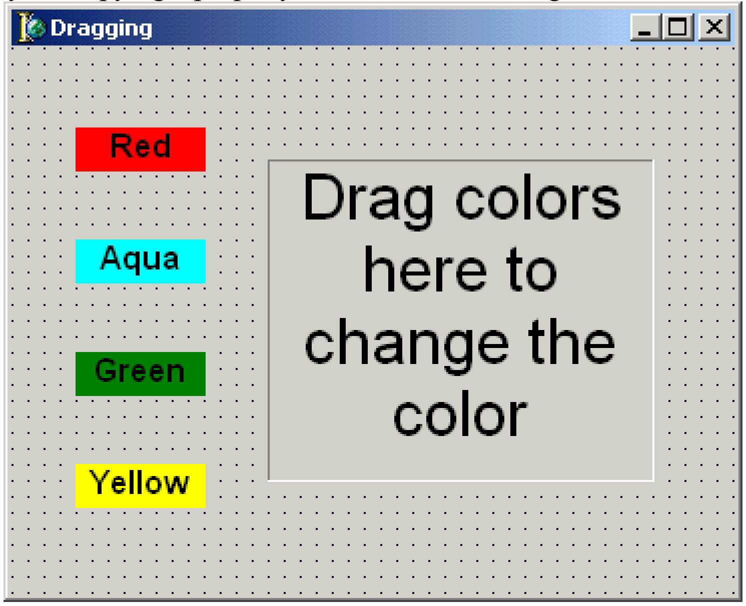
```
procedure TColorTextForm.BtnFontColorClick(Sender: TObject);
begin
  ColorDialog1.Color := Label1.Font.Color;
  if ColorDialog1.Execute then
    Label1.Font.Color := ColorDialog1.Color;
end;
```

The three lines in the body of the procedure have the following meanings: with the first, we select the background color of the label as the initial color displayed by the dialog box; with the second, we run the dialog box; with the third, the color selected by the user in the dialog box is copied back to the label. We do this last operation only if the user closes the dialog box by pressing the OK button (in which case the `Execute` method returns `True`). If the user presses the Cancel button (and `Execute` returns `False`), we skip this last statement. To change the color of the label's text, we write similar code, referring this time to the `Label1.Font.Color` property. (The complete source code for the form LabelCo example is on the CD in the file `LABELF.PAS`.) You can see the Color dialog box in action in the figure below. This dialog box can also be expanded by clicking on the Define Custom Colors button.



Dragging from One Component to Another

Before we try out other Delphi components, it's helpful to examine a particular technique: *dragging.* The dragging operation is quite simple and is increasingly common in Windows. In Windows you can drag files and programs from a folder to another, drop them on the desktop, or perform similar dragging operations on files and folders with the Windows Explorer. You usually perform this operation by pressing the mouse button on one component (or window) and releasing it on another component (or window). When this operation occurs, you can provide some code, usually for copying a property, a value, or something else to the destination component.



As an example, consider the form in the figure above. There are four color labels, with the name of each color as text, and a destination label, with some descriptive text. Actually the destination label is implemented with a StaticText component. This component has a special value for the `Border` property, `sbsSunken`, with a *lowered* effect. A similar capability is not available for plain labels. The aim of this example, named Dragging, is to be able to drag the color from one of the labels on the left to the static text, changing its color accordingly. The components have very simple properties, as the following textual description of the form summarizes:

```
object DraggingForm: TDraggingForm
  Caption = 'Dragging'
  Font.Color = clBlack
  Font.Height = -16
  Font.Name = 'Arial'
  Font.Style = [fsBold]
  object LabelRed: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'Red'
    Color = clRed
    DragMode = dmAutomatic
  end
  object LabelAqua: TLabel...
  object LabelGreen: TLabel...
  object LabelYellow: TLabel...
  object StaticTarget: TStaticText
    Alignment = taCenter
    AutoSize = False
    BorderStyle = sbsSunken
    Caption = 'Drag colors here to change the color'
    Font.Height = -32
    OnDragDrop = StaticTargetDragDrop
    OnDragOver = StaticTargetDragOver
  end
end
```

After preparing the labels by supplying the proper values for the names and caption, as well as a corresponding color, you have to enable dragging. You can do this by selecting the value `dmAutomatic` for the `DragMode` property of the four labels on the left and responding to a couple of events in the destination label.

> As an alternative to the automatic dragging mode, you might choose the manual dragging mode. This is based on the use of the `BeginDrag` and `EndDrag` methods. An alternative approach is to handle dragging manually, simply by providing a handler for events related to moving the mouse and pressing and releasing mouse buttons.

The Code for the Dragging Example

The first event I want to consider is `OnDragOver`, which is called each time you are dragging and move the cursor over a component. This event indicates that the component accepts dragging. Usually, the event takes place after a determination of whether the Source component (the one that originated the dragging operation) is of a specific type:

```
procedure TDraggingForm.StaticTargetDragOver(
  Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TLabel;
end;
```

This code accepts the dragging operation, activating the corresponding cursor, only if the Source object is really a Label component. Notice the use of the is dynamic type checking operator. The second method we have to write corresponds to the OnDragDrop event:

```
procedure TDraggingForm.StaticTargetDragDrop(
  Sender, Source: TObject;  X, Y: Integer);
begin
  StaticTarget.Color := (Source as TLabel).Color;
end;
```

To read the value of the Color property from the Source object, we need to cast this object to the proper data type, in this case TLabel. We have to perform a type conversion — technically speaking, a type downcast (a typecast from a base class to a derived class, down through the hierarchy). A type downcast is not always safe. In fact, the idea behind this cast is that we receive the parameter Source of type TObject, which is really a label, and want to use it as a TLabel object, where TLabel is a class derived fromTObject. However, in general, we face the risk of down-casting to TLabel an object that wasn't originally a label but, say, a button. When we start using the button as a label, we might have run-time errors.

In any case, when we use the as typecast, a type check is performed. Had the type of the Source object not been TLabel, an exception would have been raised. In this particular case, however, we haven't much to worry about. In fact, the OnDragDrop event is received only when the Accept parameter of the OnDragOver method is set to True, and we make this only if the Source object really is a TLabel.

Accepting Input from the User

We have seen a number of ways a user can interact with the application we write using a mouse: mouse clicks, mouse dragging, and so on. What about the keyboard? We know that the user can use the keyboard instead of the mouse to select a button by pressing the key corresponding to the underlined letter of the caption (if any).

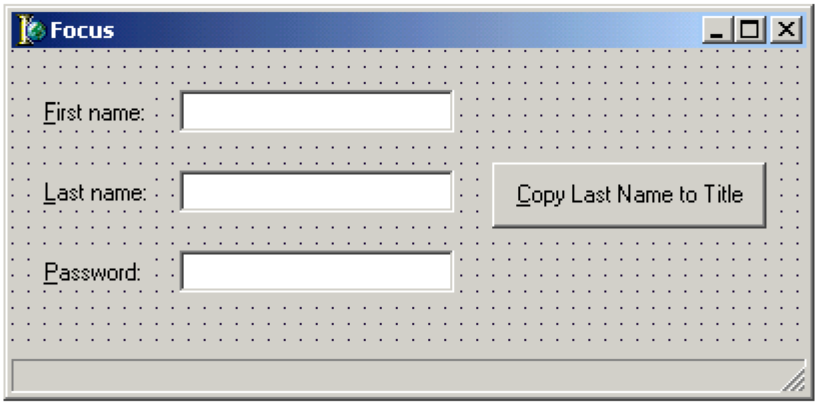
Aside from some particular cases, Windows can handle keyboard input directly. Defining handlers for keyboard-related events isn't a common operation, anyway. In fact, the system provides ready-to-use controls to build edit fields and a simple text editor. Delphi has several slightly different components in this area: Edit, MaskEdit, Memo, RichText, and the related data-aware controls. The two basic components are Edit and Memo.

An Edit component allows a single line of text and has some specific properties, such as one that allows only a limited number of characters or one that shows a special password character instead of the actual text. A Memo component, as we will see in a while, can host several lines of text.

Our first example of the Edit component, named Focus, will demonstrate a feature common to many controls, the *input focus*. In Windows, it's fairly simple to determine which is the active main window: it is in front of the other windows, and the title bar is a different color. It is not as easy to determine which window (or component) has the input focus. If the user presses a key, which component is going to receive the corresponding keyboard input message? It can be the active window, but it can also be one of its controls. Consider a form with several edit fields. Only one has the input focus at a given time. A user can move the input focus by using Tab or by clicking with the mouse on another component.

Handling the Input Focus

What's important for our example is that each time a component receives or loses the input focus, it receives a corresponding event indicating that the user either has reached (`OnEnter`) or has left (`OnExit`) the component. So we can add some methods to the form to take control over the input focus and display this information in a label or a status bar.



Besides three edit boxes, the form has also some labels indicating the meaning of the three edit fields (*First name*, *Last name*, and *Password*). For the output of the status information I've used a specific Windows common control, the StatusBar, but using a label or a panel would have had a similar effect. In fact, you can use the StatusBar component as a single-line output tool, by setting its `SimplePanel` property to `True`. Here is a summary of the properties for this example:

```
object FocusForm: TFocusForm
  Caption = 'Focus'
  object Label1: TLabel
    Caption = '&First name:'
    FocusControl = EditFirstName
  end
  object Label2: TLabel
    Caption = '&Last name:'
    FocusControl = EditLastName
  end
  object Label3: TLabel
    Caption = '&Password:'
    FocusControl = EditPassword
  end
  object EditFirstName: TEdit
    TabOrder = 0
    OnEnter = EditFirstNameEnter
  end
  object EditLastName: TEdit
    TabOrder = 1
    OnEnter = EditLastNameEnter
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    TabOrder = 2
    OnEnter = EditPasswordEnter
  end
  object ButtonCopy: TButton
    Caption = '&Copy Last Name to Title'
    TabOrder = 3
    OnClick = ButtonCopyClick
    OnEnter = ButtonCopyEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```

As you can see, the form also contains a button we can use to copy the text of the `LastNameEdit` to the form's caption. This is just an example of how to work with text entered in an edit box. As you can see in the following code, before using the `Text` property of an edit box, it's a good idea to test whether the user has actually typed something or if the edit field is still empty:

```
procedure TFocusForm.ButtonCopyClick(Sender: TObject);
begin
  if EditLastName.Text <> '' then
    FocusForm.Caption := EditLastName.Text;
end;
```

Now we can move to the most interesting part of the program. We can write a comment in the status bar each time the focus is moved to a different control.

Displaying text in the status bar as the focus moves from control to control is a good way to guide the user through the steps of an application.

To accomplish this, we need four methods, one for each of the Edit components and one for the button, referring to the `OnEnter` event. Here is the code of one of the methods (the other three event handlers are very similar):

```
procedure TFocusForm.EditFirstNameEnter(Sender: TObject);
begin
  StatusBar1.SimpleText := 'Entering the first name...';
end;
```

You can test this program with the mouse or use the keyboard. If you press the Tab key, the input focus cycles among the Edit components and the button, without involving the labels. To have a proper sequence, you can change the `TabOrder` property of the windowed component. You can change this order either by entering a proper value for this property in the Object Inspector or (much better and easier) by using the Edit Tab Order dialog box, which can be called using the Tab Order command on the form's local menu. If you open this dialog box for the Focus example. Notice that the status bar is listed but you cannot actually move onto it using the Tab key.

A second way to select a component is to use a shortcut key. It is easy to place a shortcut key on the button, but how can you jump directly to an edit box? It isn't possible directly (the `Text` of the edit box changes as a user types), but there is an indirect way. You can add the shortcut key — the ampersand (`&`) — to a label, then set the `FocusControl` property of the label to the corresponding Edit component.

> Since Windows 95, the edit controls automatically have a local menu displayed when the user presses the right mouse button over them. Although you can easily customize such a menu in Delphi (as we will see in the next chapter), it is important to realize that this is standard behavior in the system.

A Generic OnEnter Event Handler

The problem with this code is that we have to write four different `OnEnter` event handlers, copying four strings to the text of the StatusBar component. To add more edit boxes to the example, you would need to add more event handlers, copying the code over and over. And if you wanted to provide a slightly different output (for example, by changing the output of the StatusBar allowing for multiple panels), you would need to change the code many times.

The alternative solution is to write a single event handler for the `OnEnter` event of each edit box (and the button, too). We simply need to store the message for the status bar in a property, then refer to this property for the `Sender` object. A good technique is to use the `Hint` property, which is actually designed for providing descriptions to the user.

Simply store the proper messages in the `Hint` property of the edit boxes and of the button, then remove the current `OnEnter` event handler, and install this method for each of them:

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
begin
  StatusBar1.SimpleText := (Sender as TControl).Hint;
end;
```

Notice you cannot write `Sender as TEdit` because the control might be a button as well. The solution is to typecast to a common ancestor class of `TButton` and `TEdit`, which defines the `Hint` property, as you can see in the code above.

Entering Numbers

We saw in the previous example that it is very easy to use an Edit component to ask the user to input some text, although it must be limited to a single line. In general, it’s quite common to ask users for numeric input, too. To accomplish this, you can use the MaskEdit component (in the Additional page of the Components palette) or simply use an Edit component and then convert the input string into an integer, using the standard Pascal `Val` procedure or the Delphi `IntToStr` function.

This sounds good, but what if the user types a letter when a number is expected? Of course, these conversion functions return an error code, so we can use it to test whether the user has really entered a number. The second question is, when can we perform this test? Maybe when the value of the edit box changes, when the component loses focus, or when the user clicks on a particular button, such as the OK button in a dialog box. As you’ll see, not all of these techniques work well.

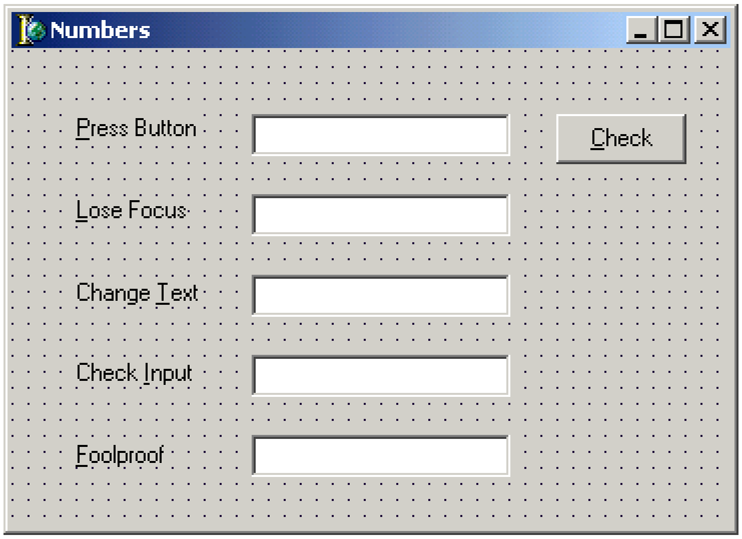
There is another, radically different, solution to the problem of allowing only numerical input in an edit box. You can look at the input stream to the edit box and stop any non-numerical input. This technique is not foolproof (a user can always paste some text into an edit box), but it works quite well and is easy to implement. Of course, you can improve it by combining it with one of the other techniques.

The next example, Numbers (see the form in the following figure), shows some of the techniques you can use to handle numerical input with an Edit component, so you can compare them easily. This example is meant as an exercise to discuss keyboard input and the input focus. To handle numerical input in an application you'll generally use specific components, as the SpinEdit or the UpDown controls available in Delphi. We will see an example of the use of another even more sophisticated control for keyboard input, the MaskEdit component.

In this example we're going to compare the effect of testing the input at different stages. First of all, build a form with five edit fields and five corresponding labels, describing in which occasion the corresponding Edit component checks the input. The form also has a button to check the contents of the first edit field. The contents of the first edit box are checked when the Check button is pressed. In the handler of the `OnClick` event of this button, the text is first converted into a number, using the `Val` procedure, which eventually returns an error code. Depending on the value of the code, a message is shown to the user:

```
procedure TNumbersForm.CheckButtonClick(Sender: TObject);
var
  Number, Code: Integer;
begin
  if Edit1.Text <> '' then
  begin
    Val (Edit1.Text, Number, Code);
    if Code <> 0 then
    begin
      Edit1.SetFocus;
      MessageDlg ('Not a number in the first edit',
        mtError, [mbOK], 0);
    end
    else
      MessageDlg ('OK, the number in the first edit box is' +
        IntToStr (Number), mtInformation, [mbOK], 0);
  end;
end;
```

If an error occurs, the application moves the focus back to the edit field before showing the error message to the user, thus inviting the user to correct the value. Of course, in this sample application a user can ignore this suggestion and move to another edit field.



The same kind of check is made on the second edit field when it loses the focus. In this case, the message is displayed automatically, but only if an error occurs. Why bother the user if everything is fine? The code here differs from that of the first edit field; it makes no reference to the `Edit2` component but always refers to the generic `Sender` control, making a safe typecast. To indicate the number of each button, I've used the `Tag` property, entering the number of the edit control.

> As you can see in the following listing, instead of casting the `Sender` parameter to the `TEdit` class several times in the same method, it is better to do this operation once, saving the value in a local variable of the `TEdit` type. The type checking involved with these casts, in fact, is quite slow.

This method is a little more complex to write, but we will be able to use it again for a different component. Here is its code:

```
procedure TNumbersForm.Edit2Exit(Sender: TObject);
var
  Number, Code: Integer;
  CurrEdit: TEdit;
begin
  CurrEdit := Sender as TEdit;
  if CurrEdit.Text <> '' then
  begin
    Val (CurrEdit.Text, Number, Code);
    if Code <> 0 then
    begin
      CurrEdit.SetFocus;
      MessageDlg ('The edit field number ' +
        IntToStr (CurrEdit.Tag) + ' does not have a valid number',
        mtError, [mbOK], 0);
    end;
  end;
end;
```

> Since Delphi 2, this code produces a warning message (a hint) when compiled, because the `Number` variable is not used after a value has been assigned to it. To avoid these hints, you can ask the compiler to disable its generation inside a specific method using the `$HINTS` compiler directive. Simply write `{$HINTS OFF}` before the method and `{$HINTS ON}` after it, as I've done in the source code of this example.

The third Edit component makes a similar test each time its content changes (using the `OnChange` event). Although we have checked the input on different occasions — using different events — the three functions are very similar to each other. The idea is to check the string once the user has entered it.

For the fourth Edit component, I want to show you a completely different technique. We are going to make a check *before* the Edit even knows that a key has been pressed. The Edit component has an event, `OnKeyPress`, that corresponds to the action of the user. We can provide a method for this event and test whether the character is a number or the Backspace key (which has a numerical value of 8, so we can refer to it as the character `#8`). If not, we change the value of the key to the null character (`#0`), so that it won't be processed by the edit control, and produce a little warning sound:

```
procedure TNumbersForm.Edit4KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

The fourth Edit component accepts only numbers for input, but it is not foolproof. A user can copy some text to the Clipboard and paste it into this Edit control with the Shift+Ins key combination (but not using Ctrl+V), avoiding any check. To solve this problem, we might think of adding a check for a change to the contents, as in the third edit field, or a check on the contents when the user leaves the edit field, as in the second component. This is the reason for the fifth, Foolproof edit field: it uses the `OnKeyPress` event of the fourth edit field, the `OnChange` method of the third, and the `OnExit` event of the second, thus requiring no new code.

To reuse an existing method for a new event, just select the Events page of the Object Inspector, move to the component, and instead of double-clicking to the left of the event name, select the button in the combo box at the right. A list of names of old methods compatible with the current event — having the same number of parameters — will be displayed.

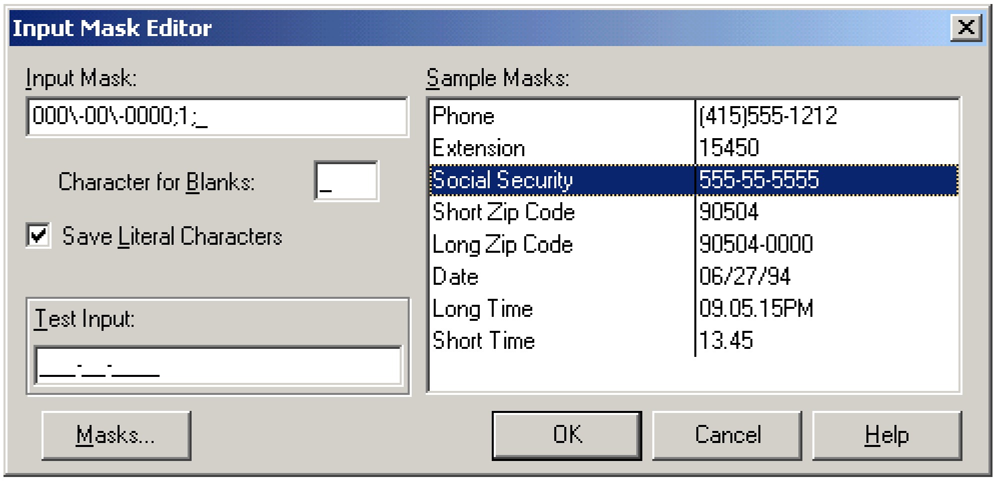
If you select the proper methods, the fifth component will combine the features of the third and the fourth. This is possible because in writing these methods, I took care to avoid any reference to the control to which they were related. The technique I used was to refer to the generic `Sender` parameter and cast it to the proper data type, which in this case was `TEdit`. As long as you connect a method of this kind to a component of the same kind, no problem should arise. Otherwise, you should make a number of type checks (using the `is` operator), which will probably make the code more complex to read. My suggestion is to share code only between controls of the same kind.

Notice also that to tell the user which edit box has incorrect text, I've added to each Edit component a value for the `Tag` property, as I mentioned before. Every edit box has a tag with its number, from 1 to 5.

Sophisticated Input Schemes

In the last example, we saw how an Edit component can be customized for special input purposes. The components could really accept only numbers, but handling complex input schemes with a similar approach is not straightforward. For this reason, Borland has supplied a ready-to-use *masked edit component*, an edit component with an input mask stored in a string.

For example, to handle numbers of no more than five digits, we can set the `EditMask` property to `99999.` (The character `9` stands for *non-compulsory digit*; refer to the Delphi documentation for the meaning of the various characters and symbols in the edit mask.) I suggest that you don't enter a string directly in this property, but instead always open the associated editor by clicking on the small ellipses button. The Input Mask editor has a test window and includes sample masks for commonly used input values.



Input Mask: 000\-00\-0000;1;_

Character for Blanks: _

☑ Save Literal Characters

Test Input: ___-__-____

Sample Masks:

Phone	(415)555-1212
Extension	15450
Social Security	555-55-5555
Short Zip Code	90504
Long Zip Code	90504-0000
Date	06/27/94
Long Time	09:05:15PM
Short Time	13:45

Notice that the Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to have the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated, by default, with semicolons).

To see more default input masks, you can press the Masks button, which allows you to open a mask file. The predefined files hold standard codes grouped by country. For example, if you open the Italian group, you can find the taxpayer number (or *fiscal code*, which is used like social security numbers in the U.S.). This code is a complex mix of letters and numbers (including the consonants representing name, birth date, area code, and more), as its mask demonstrates:

```
LLLLLL00L00L000L
```

In this kind of code, `L` stands for a letter and `0` for a number. While you can look these up in the Help file, there is a summary of these codes in the following Mask1 example.

The form of this example includes a MaskEdit and an Edit component. The Edit is used to change the `EditMask` property of the first one at run-time. To accomplish this, I've just written a couple of lines of code to copy the text of the property into the edit box at the beginning (the `OnCreate` event) and reverse the action each time the plain edit box changes (`Edit1Change`):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.Text := MaskEdit1.EditMask;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  MaskEdit1.EditMask := Edit1.Text;
end;
```

The form also has a list box with the description of the most important codes used to build the mask.

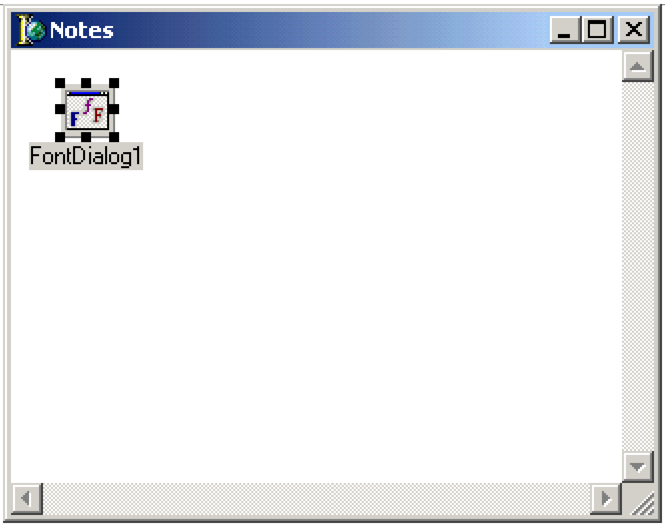
Creating a Simple Editor

Edit components can handle a limited amount of text, and only on a single line. If you need to accept longer text input, you should use the Memo component. A Memo component is like an Edit component, but it can span several lines, contain scroll bars to move through the text, and contain more text.

The easiest way to use a Memo is as a text editor, as you'll see in the next example, Notes. The idea is to implement an editor covering all of the window (or form) which contains it, to resemble Windows Notepad. The only other feature we will implement is to give the user the option of choosing the font for the editor.

Both parts are very easy to implement. Create a new project and place a Memo component on the form. Delete its text, remove the border, and set the `Alignment` property to `alClient`, so that it will always cover the whole client area — the internal surface — of the form. Also add both scroll bars, horizontal and vertical, selecting the value `ssBoth` for the memo's `ScrollBars` property. Here is the summary (and the design time image of the form):

```
object NotesForm: TNotesForm
  Caption = 'Notes'
  object Memo1: TMemo
    Align = alClient
    BorderStyle = bsNone
    Font.Height = -19
    Font.Name = 'Times New Roman'
    ScrollBars = ssBoth
    OnDblClick = Memo1DblClick
  end
  object FontDialog1: TFontDialog...
end
```



The Font Dialog Box

The second portion of the program involves the font. In the same way we used the standard Color dialog box in a previous example, we can use the standard Font selection dialog box provided by Windows. Just move to the Dialogs page of the Components palette and select the FontDialog component. Place it anywhere on the form, and add the following code when the user double-clicks inside the Memo:

```
procedure TNotesForm.Memo1DblClick(Sender: TObject);
begin
  FontDialog1.Font := Memo1.Font;
  if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;
```

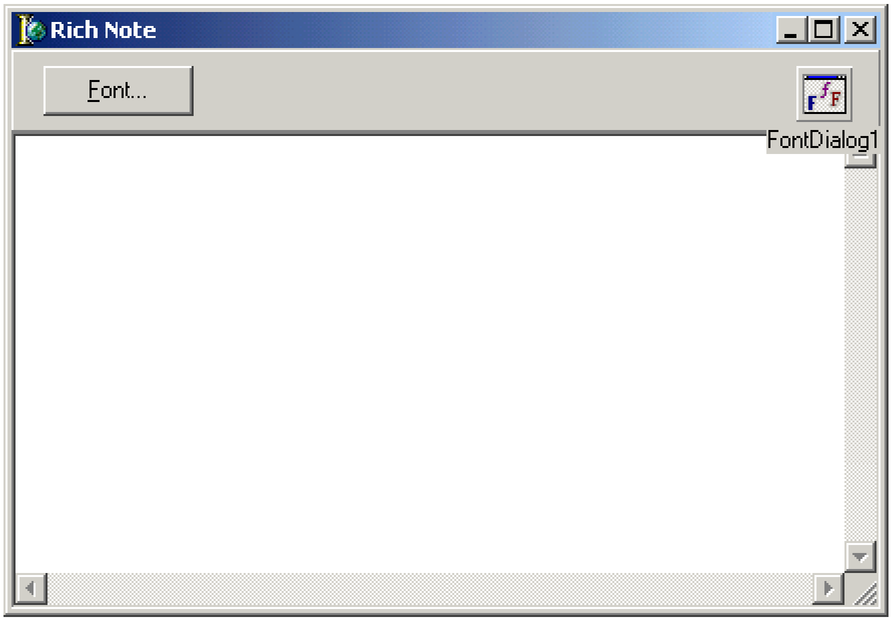
This code copies the current font to the corresponding property of the dialog component so it will be selected by default. Then it executes the dialog box. At the end, the `Font` property will contain the font the user selected. If the user presses the OK button, the third line of the above code copies the font back to the Memo.

This program is more powerful than it appears at first glance. For example, it allows copy and paste operations using the keyboard — this means you can copy text from your favorite word processor — and can handle the color of the font. Why not use it to place a big and colorful message on your screen?

Creating a Rich Editor

Although you can choose a font in the Notes program, all of the text you have written will have the same font. Windows has a control that can handle the Rich Text Format (RTF). A Delphi component, RichEdit, encapsulates the behavior of this standard control.

You can find an example of a complete editor based on the RichEdit component among the examples that ship with Delphi. (The example is named RichEdit, too). Here, we'll only change the previous program slightly by replacing the Memo component with a RichEdit, and allow a user to change the font of the selected portion of the text, not the whole text.



The RichNote example has a RichEdit component filling its client area. However, the component has no double-click event, so I added a button to select the font and placed it in a panel aligned to the top of the form, making a very simple toolbar. Here is the textual description of some of the properties of the three components:

```
object RichEdit1: TRichEdit
  Align = alClient
  HideScrollBars = False
  ScrollBars = ssBoth
end
object Panel1: TPanel
  Align = alTop
  object Button1: TButton
    Caption = '&Font...'
  end
end
```

Notice the caption of the button, which has an ampersand for the shortcut key, and an ellipsis at the end to indicate that pressing it will open a dialog box. When the user clicks on the button, if some text is selected, the program shows the standard Font dialog box using the default font of the RichEdit component as the initial value. At the end, the selected font is copied to the attributes of the current selection. The `DefAttributes` and `SelAttributes` properties of the RichEdit component are not of the `TFont` type, but they are compatible, so we can use the `Assign` method to copy the value:

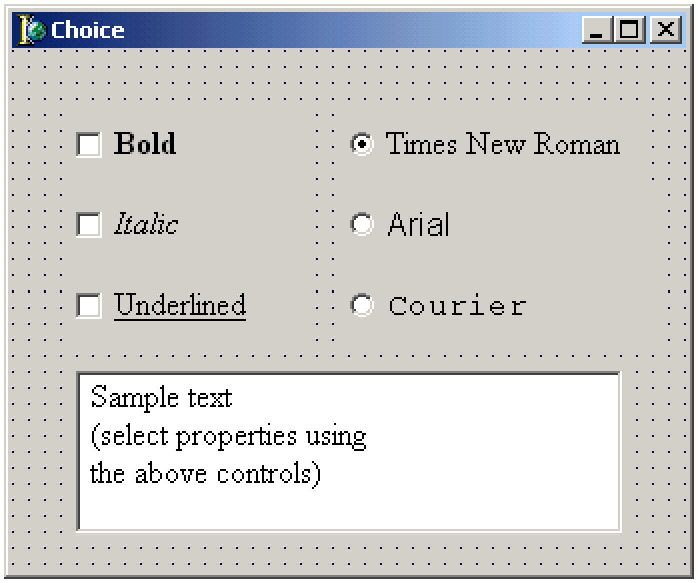
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign(RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign(FontDialog1.Font);
  end
  else
    ShowMessage ('No text selected');
end;
```

The RichEdit component has other attributes related to fonts and paragraph formatting. We will use this component in further examples of the book; however, the simple code above is enough to let users produce much more complex output than the Memo component allows.

Making Choices

There are two standard Windows controls that allow the user to choose different options. The first is the *check box*, which corresponds to an option that can be selected freely (unless it has been disabled). The second control is the *radio button*, which corresponds to an exclusive selection. For example, if you see two radio buttons with the labels *A* and *B*, you can select *A* or select *B*, but not both of them at the same time. The other characteristic of a multiple choice is that you *must* check one of the radio buttons.

If the difference between check boxes and radio buttons is still not clear, an example might help you. The example has three check boxes to select the style *Bold*, *Italic*, or *Underlined*, and three radio buttons to choose a font (*Times New Roman*, *Arial*, or *Courier*). There is also a memo field with some text to show the effect of the user selections immediately.



Sample text
(select properties using
the above controls)

The difference between the use of the check boxes and the radio buttons should be obvious. The text might be bold and italic at the same time, but it cannot be Arial and Courier at once. A user must choose only one font (and cannot choose none) but can select each of the styles independently from the other two (including no style at all).

This program requires some simple code. Each time the user clicks on a check box or radio button, we have to create a corresponding action. For the text styles, we have to look at the `Check` property of the control and add or remove the corresponding element from the memo's Font property `Style` set:

```
procedure TForm1.CheckBoldClick(Sender: TObject);
begin
  if CheckBold.Checked then
    Memo1.Font.Style := Memo1.Font.Style + [fsBold]
  else
```

```
    Memo1.Font.Style := Memo1.Font.Style - [fsBold];
end;
```

The other two check boxes have similar code for their `OnClick` events. The basic code for the radio buttons is even simpler since you cannot deselect a radio button by clicking on it:

```
procedure TForm1.RadioTimesClick(Sender: TObject);
begin
  Memo1.Font.Name := 'Times New Roman';
end;
```

Grouping Radio Buttons

Radio buttons represent exclusive choices. However, a form might contain several groups of radio buttons. Windows cannot determine by itself how the various radio buttons relate to each other. The solution, both in Windows and in Delphi, is to place the related radio buttons inside a container component. The standard Windows user interface uses a group box control to hold the radio buttons together, both functionally and visually. In Delphi, this control is implemented in the GroupBox component. However, Delphi has a second, similar component that can be used specifically for radio buttons: the RadioGroup component. A RadioGroup is a group box with some radio button inside it.

Using the radio group is probably easier than using the group box, but I'll use the more traditional approach to show you the code you can write to work with controls that have been placed inside another control. The RadioGroup component can automatically align its own internal radio buttons, and you can easily add new choices at run-time. You can see the differences between the two approaches in the next example.

The rules for building a group box with radio buttons are very simple. Place the GroupBox component in the form, then place the radio buttons in the group box. The GroupBox component contains other controls and is one of the container components used most often, together with the Panel component. If you disable or hide the group box, all the controls inside it will be disabled or hidden.

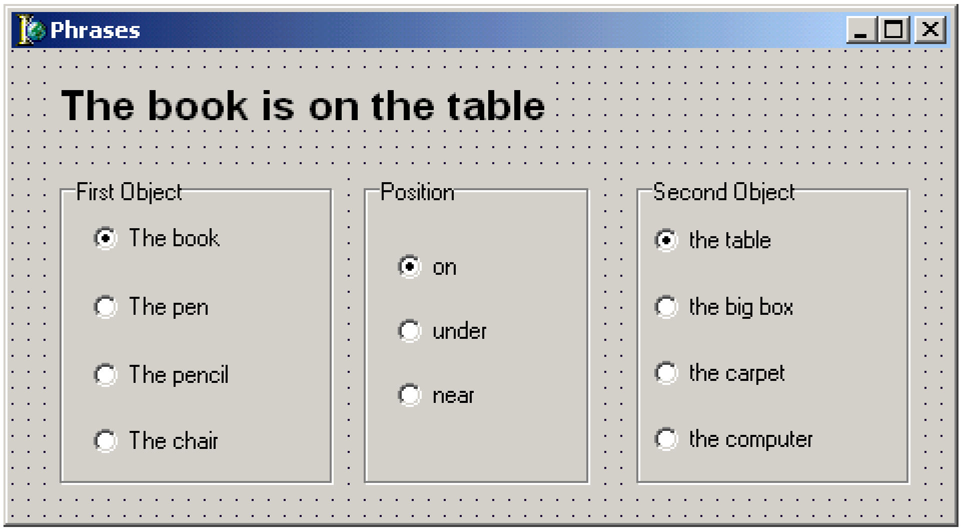
You can continue handling the individual radio buttons, but you might as well navigate through the array of controls owned by the group box. As discussed in the last chapter, the name of this property referring to this array of controls is `Controls`. Another property, `ControlCount`, holds the number of elements. These two properties can be accessed only at run-time.

The Phrases1 Example

If you've ever tried to learn a foreign language, you probably spent some time repeating the same silly and useless phrases over and over. Probably the most typical, when you learn English, is the infamous *"The book is on the table."* To demonstrate radio buttons, the Phrases1 example creates a tool to build such phrases by choosing among different available options.

This form is quite complex. If you rebuild it, remember that you must place the GroupBox components first and the radio buttons later. After doing this, you have to enter a proper caption for each element. The last selection is based on a radio group component, instead of a group box holding some radio buttons (as you can see in the textual description of the form below). In this case you create the options by entering a list of values in the Items property.

Remember that you also need to add a label, select a large font for it, and enter text corresponding to the radio buttons that are checked at design-time. This is an important point: When you place some radio buttons in a form or in a group box, remember to check one of the elements at design-time. One radio button in each group should always be checked, and the `ItemIndex` property of the radio group, indicating the current selection, should have a proper value.



Here is the textual description of the form with a summary of this information:

```
object Form1: TForm1
  Caption = 'Phrases'
  object Label1: TLabel
    Width = 243
    Caption = 'The book is on the table'
    Font.Height = -21
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object GroupBox1: TGroupBox
    Caption = 'First Object'
    object RadioBook: TRadioButton
      Caption = 'The book'
      Checked = True
      OnClick = ChangeText
    end
    object RadioPen: TRadioButton
      Caption = 'The pen'
      OnClick = ChangeText
    end
    object RadioPencil: TRadioButton...
    object RadioChair: TRadioButton...
  end
  object GroupBox2: TGroupBox
    Caption = 'Position'
    object RadioOn: TRadioButton
      Caption = 'on'
      Checked = True
      OnClick = ChangeText
    end
```

```
      object RadioUnder: TRadioButton...
      object RadioNear: TRadioButton...
    end
    object RadioGroup1: TRadioGroup
      Caption = 'Second Object'
      Items.Strings = (
        'the table'
        'the big box'
        'the carpet'
        'the computer')
      OnClick = ChangeText
    end
  end
```

Now we have to write some code so that when the user clicks on the radio buttons, the phrase changes accordingly. There are different ways to do this. One is to follow the same approach as in the last example, providing a method for each button's `OnClick` event. Then we need to store the various portions of the phrase in some of the form's variables, change the portion corresponding to that button, and rebuild the whole phrase.

An alternative solution is to write a single method that looks at which buttons are currently checked and builds the corresponding phrase. This single method must be connected to the `OnClick` event of every radio button and of the RadioGroup component, a task we can easily accomplish. Select each of the radio buttons on the form (clicking on each one while you hold down the Shift key) and enter the name of the method in the Object Inspector. Since the method used to compute the new phrase doesn't refer to a specific control, you might name it yourself, simply entering a name in the second column of the Object Inspector next to the `OnClick` event. Here is the code of this single complex method:

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: string;
  I: integer;
begin
  {look at which radio button is selected
  and add its text to the phrase}
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Phrase := (GroupBox1.Controls[I] as TRadioButton).Caption;

  {add the verb and blank spaces}
  Phrase := Phrase + ' is ';

  {repeat the operation on the second group box}
  for I := 0 to GroupBox2.ControlCount - 1 do
    with GroupBox2.Controls[I] as TRadioButton do
      if Checked then
        Phrase := Phrase + Caption;

  {retrieve the radio group selection, and display
  the result in the label}
  Label1.Caption := Phrase + ' ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex];
end;
```

The `ChangeText` method starts looking at which of the first group of radio buttons is selected, then moves on to adding a verb and the proper spaces between words. To determine which control in a group box is checked, the procedure scans these controls in a `for` loop. The `for` loop ranges from 0 to the number of controls minus 1, because the `Controls` array is zero-based, and tests whether the `Checked` property of the radio button is `True`. A cast is required to perform this operation — we cannot use the `Checked` property on a generic control. When the checked radio button has been found, the program simply copies its caption to the string. At this point, the `for` loop might terminate, but since only one radio button is checked at a time, it is safe to let it reach its natural end — testing all the elements. The same operation is repeated two times, but you can see that the second time a `with` statement is used to make the code shorter and more readable.

As you can see from the final portion of the method above, if you are using the RadioGroup component, the code is much simpler. This control, in fact, has an `ItemIndex` property indicating which radio button is selected and an `Items` property with a list of the text of the fake radio buttons. Overall, using a radio group is very similar to using a list box (as we will see in the next example), aside from the obvious difference in the user interface of the two components.

A List with Many Choices

If you want to add many selections, radio buttons are not appropriate, unless you create a really big form. The usual number of radio buttons is no more than 5 or 6. Another problem is that although you can disable a radio button, the elements of a group are usually fixed. Only when using a radio group can you have some flexibility. For both of these problems, the solution is to use a list box. A list box can host a large number of choices in a small space, because it can contain a scroll bar to show on screen only a limited portion of the whole list. Another advantage of a list box is that you can easily add new items to it or remove some of the current items. List boxes are extremely flexible and powerful.

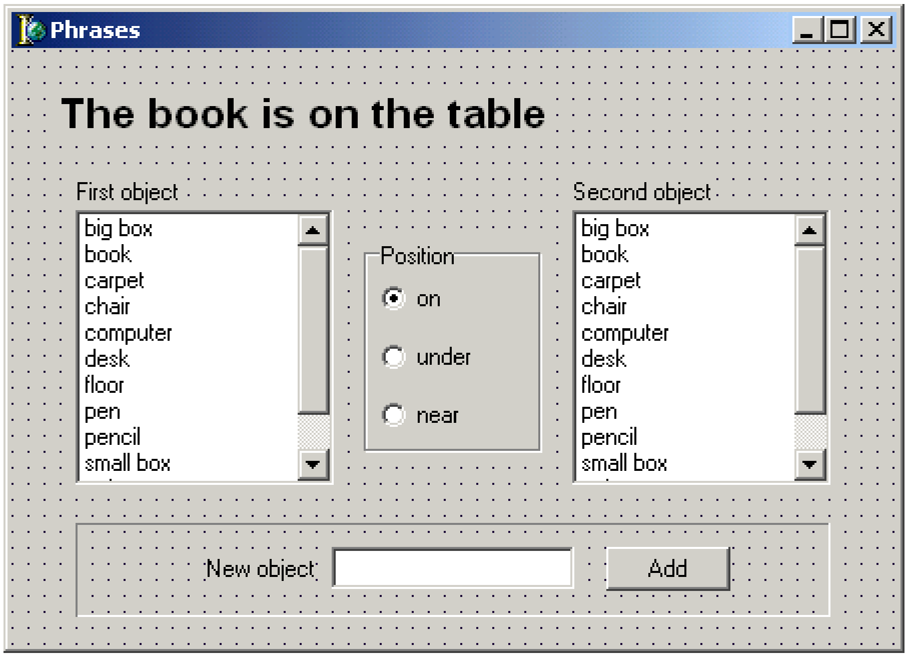
> Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, a behavior similar to a group of radio buttons, and allowing multiple selections, which is similar to a group of check boxes. The next version of this example will have a multiple-selection list box.

For the moment, let's focus on a single-selection list box. We might use a couple of these components to change the Phrases1 example slightly. Instead of having a number of radio buttons to select the first and second objects of the phrase, we can use two list boxes. Besides allowing us to have a larger number of items, the advantage is that we can allow the user to insert new objects in the list and prevent selection of the same object twice, to avoid a phrase such as "The book is on the book." As you might imagine, this example is really much more complicated than the previous one and will require some fairly complex code.

The Form of the Phrases2 Example

As usual, the first step is to build a form. You can start with the form from the last example and remove the two group boxes on the sides and replace them with two list boxes. The radio buttons inside the group boxes will be deleted automatically. I've also replaced the central group box with a radio group. Actually, there's not much left hfrom the previous example!

Now, add some strings to the `Items` property of both list boxes. For the example to work properly, the two list boxes should have the same strings; you can copy and paste them from the editor of the `Items` property of one list box to the editor of the same property of the other component. To improve the usability of the program, you might sort the strings in the list boxes, setting their `Sorted` property to `True`. Remember also to add a couple of labels above the list boxes, to describe their contents.



In the lower part of the form, I've also added an edit field, with its label, and a button, and a bevel around them to group them visually (the bevel is just a graphical component, not a container). As we will see later, when a user presses the button, the text in the Edit control is added to both list boxes. This operation will take place only if the text of the edit box is not empty and the string is not already present in the list boxes.

Here is the textual description of the components of this updated form (which is really very different from the previous version):

```
object Form1: TForm1
  Caption = 'Phrases'
  OnCreate = FormCreate
  object Label1: TLabel... // as in Phrases1
  object Label2: TLabel
    Caption = 'First object'
  end
  object Label3: TLabel
    Caption = 'Second object'
  end
  object ListBox1: TListBox
    Items.Strings = (
      'big box'
      'book'
      'carpet'
      'chair'
      'computer'...)
    Sorted = True
    OnClick = ChangeText
  end
  object RadioGroup1: TRadioGroup
    Caption = 'Position'
    Items.Strings = (
      'on'
      'under'
      'near')
  end
  object ListBox2: TListBox... // identical to ListBox1
  object Bevel1: TBevel...
  object Label4: TLabel
    Caption = 'New object'
  end
  object EditNew: TEdit...
  object ButtonAdd: TButton
    Caption = 'Add'
    OnClick = ButtonAddClick
  end
end
```

Working with the List Boxes

Once you have built this or a similar form, you can start writing some code. The first thing to do is to provide a new `ChangeText` procedure, connected with the `OnClick` event of the radio group and of the two list boxes. This procedure is simpler than in the previous example. In fact, to retrieve the selected text from the list box, you only need to get the number of the item selected (stored in the run-time property `ItemIndex`) and then retrieve the string at the corresponding position of the Item array, as the Phrases1 program did.

Here is what the code for the procedure looks like initially (this is just a temporary version, different from the one in the source code):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase: String;
begin
  Phrase := 'The ';
  Phrase := Phrase + ListBox1.Items [ListBox1.ItemIndex];
  Phrase := Phrase + ' is ';
  Phrase := Phrase + RadioGroup1.Items [RadioGroup1.ItemIndex];
  Phrase := Phrase + ' the ';
  Phrase := Phrase + ListBox2.Items [ListBox2.ItemIndex];
  Label1.Caption := Phrase;
end;
```

This program, however, won’t work properly because, at the beginning, no item is selected in either list box. To solve this problem, we can add some code to the form’s `OnCreate` event. In this code, we can look for the two default strings, book and table, and select them. You should do this operation in two steps. First, you need to look for the string’s index in the array of strings, with the `IndexOf` method. Then you can use that value as the index of the currently selected item:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
```

```
  N := ListBox1.Items.IndexOf ('book');
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf ('table');
  ListBox2.ItemIndex := N;
end;
```

Removing a Selected String from the Other List Box

Once this part of the program works, we have two more problems to solve: We must remove the selected string from the other list box (to avoid using the same term twice in a phrase), and we must write the code for the click event on the button.

The first problem is more complex, but I'll address it immediately since the solution of the second problem will be based partially on the code we write for the first one. Our aim is to delete from a list box the item currently selected in the other list box. This is easy to code. The problem is that once the selection changes, we have to restore the previous items, or our list boxes will rapidly become empty. A good solution is to store the two currently selected strings for the two list boxes in two private fields of the form, `String1` and `String2`:

```
type
  TForm1 = class(TForm)
  ...
 private
    String1, String2: String;
  end;
```

Now we have to change the code executed at startup and the code executed each time a new selection is made. In the `FormCreate` method, we need to store the initial value of the two strings and remove them from the other list box; the first string should be removed from the second list box, and vice versa. Since the `Delete` method of the `TStrings` class requires the index, we have to use the `IndexOf` function again to determine it:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  N : Integer;
begin
  String1 := 'book';
  String2 := 'table';

  {delete the selected string from the other list box
  to avoid a double selection}
  ListBox2.Items.Delete (ListBox2.Items.IndexOf (String1));
  ListBox1.Items.Delete (ListBox1.Items.IndexOf (String2));

  {select the two strings in their respective list boxes}
  N := ListBox1.Items.IndexOf (String1);
  ListBox1.ItemIndex := N;
  N := ListBox2.Items.IndexOf (String2);
  ListBox2.ItemIndexh := N;
end;
```

> The code to select the string should be executed after calling `Delete`, because removing an element before the one currently selected will alter the selection. The fact is that the selection is just a number referring to a string, not the reverse, as it should probably be. By the way, this doesn't depend on Delphi implementation but on the behavior of list boxes in Windows.

Things get complicated when a new item is selected in one of the list boxes. The `ChangeText` procedure has some new code at the beginning, executed only if the click took place on one of the list boxes (remember that the code is also associated with the group box). For each string, we have to check whether the selected item has changed and, in this case, add the previously selected string to the other list box and delete the new string. Here is the new version of the `ChangeText` method:

```
procedure TForm1.ChangeText(Sender: TObject);
var
  TmpStr: String;
begin
  // if a list box has changed
  if Sender is TListBox then
  begin
    // get the text of the first string
    TmpStr := ListBox1.Items [ListBox1.ItemIndex];
    // if the first one has changed
    if TmpStr <> String1 then
    begin
      // update the strings in ListBox2
      {1.} ListBox2.Items.Add (String1);
      {2.} ListBox2.Items.Delete (
             ListBox2.Items.IndexOf (TmpStr));
      {3.} ListBox2.ItemIndex :=
             ListBox2.Items.IndexOf (String2);
      {4.} String1 := TmpStr;
    end;

    // get the text of the second string
    TmpStr := ListBox2.Items [ListBox2.ItemIndex];
    // if the second one has changed
    if TmpStr <> String2 then
    begin
      // update the strings in ListBox1
      ListBox1.Items.Add (String2);
      ListBox1.Items.Delete (ListBox1.Items.IndexOf (TmpStr));
      ListBox1.ItemIndex := ListBox1.Items.IndexOf (String1);
      String2 := TmpStr;
    end;
  end;

  // build the phrase with the current strings
  Label1.Caption := 'The ' + String1 + ' is ' +
    RadioGroup1.Items [RadioGroup1.ItemIndex] +
    ' the ' + String2;
end;
```

What is the effect of the first part of this code? Here is a detailed description of the operations, referring to a new selection in the first list box. The procedure stores the selected element of the first list box in the temporary string TmpStr. If this is different from the older selection, String1, four operations take place (refer to the numbers in the listing above):

1. The previously selected string, String1, is added to the other list box, ListBox2.
2. The new selection, TmpStr, is removed from the other list box.
3. The selected string of the other list box, String2, is reselected in case its position has been changed by the two preceding operations.
4. Once the two lists contain the correct elements, we can store the new value in String1 and use it later on to build the phrase.

We perform the same steps for the other list box a few lines later. Notice that we don't need to access the list boxes again to build the phrase at the end of the OnChange method, since String1 and String2 already contain the values we need.

Implementing the OnClick event for the Add button is quite simple. The only precautions we have to take are to test whether there is actually some text in the edit box or if it is empty and to check whether the string is already present in one of the two list boxes. Checking only one of the list boxes will miss a correspondence between the text of the edit box and the item currently selected in the other list box.

To make this check, we can ask both ListBox components for the index of the new string; if it is not present in the list — if there is no match — the value –1 will be returned. Otherwise, the IndexOf function returns the correct index, starting with 0 for the first element. In technical terms, we can say that the function returns the zero-based index of the element, or the error code –1 if it is not found. Here is the code:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  {if there is a string in the edit control and
  the string is not already present in one of the lists}
  if (EditNew.Text <> '') and
    (ListBox1.Items.IndexOf(EditNew.Text) < 0) and
    (ListBox2.Items.IndexOf(EditNew.Text) < 0) then
  begin
    {add the string to both list boxes}
    ListBox1.Items.Add (EditNew.Text);
    ListBox2.Items.Add (EditNew.Text);

    {reselects the current items properly}
    ListBox1.ItemIndex := ListBox1.Items.IndexOf (String1);
    ListBox2.ItemIndex := ListBox2.Items.IndexOf (String2);
  end
  else
    MessageDlg ('The edit control is empty or contains'
      + ' a string which is already present',
      mtError, [mbOK], 0);
end;
```

In the final part of this method's code, we need to reselect the current item of each list box since the position of the selected item might change. This happens if the new item is inserted before the one that is currently selected — that is, if it has a lower sort order.

Allowing Multiple Selections

A list box can allow the selection of either a single element or a number of elements. We make this choice in setting up a list box by specifying the value of its `Multiple` property. As the name implies, setting `Multiple` to `True` allows multiple selections. There are really two different kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case a user selects multiple items simply by clicking on them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items. This second choice is determined by the `ExtendedSelect` property.

While setting up a multiple-selection list box is very simple, the problems start to appear when you have to write the code. Accessing the selected item of a single-selection list box is simple. The `ItemIndex` property holds the index of the selected item, and the selected string can be retrieved with a simple expression:

```
ListBox2.Items[ListBox2.ItemIndex];
```

In a multiple-selection list box, on the other hand, we do not know how many items are selected, or even whether there is any item selected. In fact, a user can click on an item to select it, drag the cursor to select a number of consecutive items in the list, or click the mouse button on an item while holding down Ctrl key to toggle the selection of a single item without affecting the others. Using this last option, a user can even deselect all the items in a list box.

A program can retrieve information on the currently selected items by examining the `Selected` array. This array of `Boolean` values has the same number of entries as the list box. Each entry indicates whether the corresponding item is selected.

For example, to know how many items are selected in a list box, we need to scan the `Selected` array, usually with a `for` loop ranging from `0` to the number of items in the list minus one:

```
SelectCount := 0;
for ListItem := 0 to ListBox1.Items.Count - 1 do
  if ListBox1.Selected[ListItem] then
    Inc (SelectCount);
```

Actually, the ListBox component has an undocumented `SelCount` property, you can use to obtain exactly the information computed in the code above. We won't use this code in the next example, anyway, but a more complex version.

The Third Version of the Phrases Example

With this information, we can build a new version of the Phrases example, allowing a user to select several items in the first list box. The only real difference between the form of this new version and that of the last one is the value of the `MultiSelect` property in the first list box is not set to `True`.

In addition, the label at the top of the form has been enlarged, enabling the `WordWrap` property and disabling the `AutoSize` property, to accommodate longer phrases. Since the example's code is complex enough, I've removed the portion used to delete from a list box the item selected in the other list box. In the case of multiple selections, this would have been really complicated.

The main problem we face is building the different phrases correctly. The basic idea is to scan the `Selected` array each time and add each of the selected objects to the phrase. However, we need to place an *"and"* before the name of the last object, omitting the comma if there are only two. Moreover, we need to decide between singular and plural (*is* or *are*) and provide some default text if no element is selected. As you can see from the table below, building these phrases is not simple. In fact, if we store the phrase *"The book and the computer,"* when we need to add a third item, we must go back and change it.

Items Selected	SelectCount	Phrase
{none}	0	Nothing is
book	1	The book is
book, computer	2	The book and the computer are
book, computer, pen	3	The book, the computer, and the pen are
book, computer, pen, small box	4	The book, the computer, the pen, and the small box are

An alternative idea is to create two different phrases, one valid if no other elements will be added, the other prepared to host future objects (without the *and*). In the code, the `TmpStr1` string is the tentative final statement, while `TmpStr2` is the temporary string used to add a further element. At the end of the loop, `TmpStr1` holds the correct value. As you can see in the code below, in case only two items are selected we have to build the phrase in a slightly different way, removing the comma before the *and* conjunction.

Notice when scanning a sorted list box that the objects are always added to the resulting string in alphabetical order, not in the order in which they were selected. You can study how this idea has been implemented by looking at the new version of the `ChangeText` method, in the following code (and by studying the following table, which describes step-by-step how the strings are built):

```
procedure TForm1.ChangeText(Sender: TObject);
var
  Phrase, TmpStr1, TmpStr2: String;
  SelectCount, ListItem: Integer;
begin
  SelectCount := 0;

  {look at each item of the multiple selection list box}
  for ListItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [ListItem] then
    begin
      {if the item is selected increase the count}
      Inc (SelectCount);
      if SelectCount = 1 then
      begin
        {store the string of the first selection}
        TmpStr1 := ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr1;
      end
      else if SelectCount = 2 then
      begin
        {add the string of the second selection}
        TmpStr1 := TmpStr1 + ' and the ' +
          ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr2 + ', the ' +
          ListBox1.Items.Strings [ListItem];
      end
      else // SelectCount > 2
      begin
```

```
        {add the string of the further selection}
        TmpStr1 := TmpStr2 + ', and the ' +
          ListBox1.Items.Strings [ListItem];
        TmpStr2 := TmpStr2 + ', the ' +
          ListBox1.Items.Strings [ListItem];
      end;
    end;

  {build the first part of the phrase}
  if SelectCount > 0 then
    Phrase := 'The ' + TmpStr1
  else
    Phrase := 'Nothing';

  if SelectCount <= 1 then
    Phrase := Phrase + ' is '
  else
    Phrase := Phrase + ' are ';

  {add the text of the radio button}
  Phrase := Phrase +
    RadioGroup1.Items [RadioGroup1.ItemIndex];

  {add the text of the second list box}
  Phrase := Phrase + ' the ' +
    ListBox2.Items [ListBox2.ItemIndex];
  Label1.Caption := Phrase;
end;
```

Items Selected	Steps	TmpStr1 (Tentative Final Statement)	TmpStr2 (Temporary Statement)
book	1	book	book
+ computer	2	book and the computer	book, the computer
+ pen	3	book, the computer, and the pen	book, the computer, the pen
+ small box	4	book, the computer, the pen, and the small box	book, the computer, the pen, the small box

The other procedures of the program change only slightly. The `FormCreate` method is simplified because we do not need to delete the selected item from the other list box. The `Add` method is simplified because both list boxes always have the same items and because the multiple-selection list box creates no problems with the selection if you add a new element.

An alternative solution to handle the status of multiple-selection list boxes is to look at the value of the `ItemIndex` property, which holds the number of the item of the list having the focus. If a user clicks on several items while holding down Ctrl, each time a click event takes place, you know which of the items have been selected or deselected — you can easily determine which of the two operations took place by looking at the value of the `Selected` array for that index. The problem is that if the user selects a number of elements by dragging the mouse, this method won't work. You need to intercept the dragging events, and this is considerably more complex than the technique described earlier.

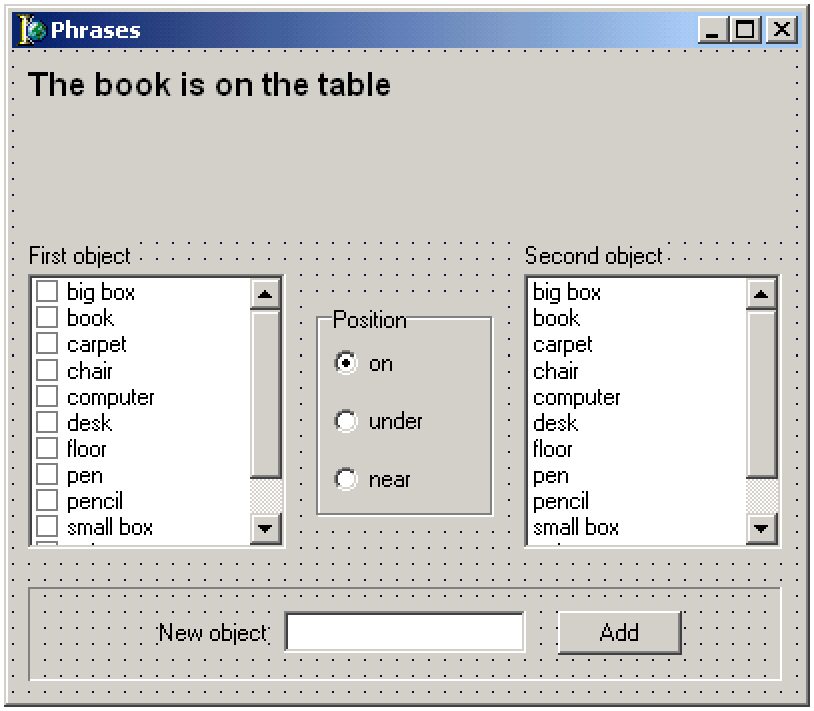
Using a CheckListBox Component

A further extension to the Phrases example is the use of the CheckListBox component, a component originally introduced by Borland in Delphi 3. This is basically a list box with a custom output (or an *owner-draw* list box, to use the proper technical term). Each item of the list is preceded by a check box. A user can select a single item of the list, but can also click on the check boxes to toggle their status.

If the component has the `AllowGrayed` property set to `True`, then each check box can be non-selected, grayed, or selected. Clicking on the check box alternates these three possible conditions. To check the current status of each item you can use the `Checked` and the `State` property. Both are array properties. The first, `Checked`, is a Boolean property you should use when `AllowGrayed` is set to `False`. The second, `State`, is a property of the `TCheckBoxState` data type:

```
type
  TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed)
```

This property should be used when `AllowGrayed` is set to `True`, to distinguish among the three different states of each item. Apart from these properties, the specific user interface, and the new `OnClickCheck` event, this component behaves as a ListBox.



As I mentioned at the beginning of this section, to show you an example of the use of this component I've further updated the Phrases3 example, building the Phrases4 version. I've basically replaced the first multiple selection list box with the new CheckListBox component, set its `Sorted` property to `True`, and copied the `Items`. Then I've updated the code, replacing the `ListBox1` object with the `CheckListBox1` object, and replacing the `Selected` property of the first with the `Checked` property of the second in the `ChangeText` method. Here is an excerpt of the new version of this method:

```
  for ListItem := 0 to CheckListBox1.Items.Count - 1 do
```

```
if CheckListBox1.Checked [ListItem] then
begin
  {if the item is selected increase the count}
  Inc (SelectCount);
  if SelectCount = 1 then
  begin
    {store the string of the first selection}
    TmpStr1 := CheckListBox1.Items.Strings [ListItem];
    TmpStr2 := TmpStr1;
  end
  else if SelectCount = 2 then
    ...
```

The important point of this program is that this component makes it more obvious to the user that the list box allows multiple selections. The plain list box, in fact, gives no clue of this fact.

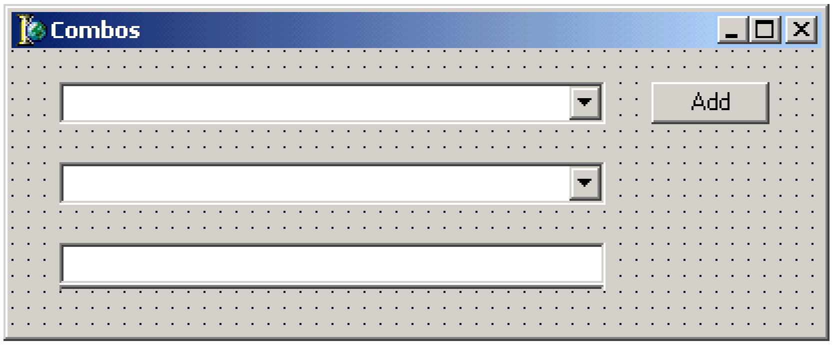
Many Lists, Little Space

List boxes take up a lot of screen space, and they offer a fixed selection. That is, a user can choose only among the items in the list box and cannot make any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control. A combo box is similar to an edit box, and you can often enter some text in it. It is also similar to a list box, with a drop-down arrow that displays a list box. Even the name of the control suggests that it is a combination of two other controls, an Edit and a ListBox. However, the behavior of a ComboBox component might change a lot, depending on the value of its `Style` property. Here is a short description of the various styles:

- The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request.
- The `csDropDownList` style defines a combo box that does not allow editing. By pressing a key, the user selects the first word starting with that letter in the list.
- The `csSimple` style defines a combo box that always displays the list box below it. This version of the control allows direct editing.
- The `csOwnerDrawFixed` and `csOwnerDrawVariable` styles define combo boxes based on an owner-draw list — that is, a list containing graphics determined by the program rather than simple strings.

To see the difference between the first three types, you can run the Combos example, which I'll describe in a moment. As you can appreciate by testing the program, Combos displays three combo boxes having three different styles: drop-down, drop-down list, and simple.



This program is very simple. Each combo box has the same basic strings—the names of more than 20 different animals. The first combo box contains an *Add* button. If the user presses the button, any text entered in the combo box is added to its list, provided it is not already present. This is the code associated with the `OnClick` event of the button:

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
with ComboBox1 do
  if (Text <> '') and (Items.IndexOf (Text) < 0) then
    Items.Add (Text);
end;
```

You can use the second combo box to experiment with the automatic lookup technique. If you press a key, the first of the names in the list starting with that letter will be selected. By pressing the up arrow key and the down arrow key, you can further navigate in the list without opening it. This navigation technique of using initial letters and arrows can be used with each of the combo boxes.

The third combo box is a variation of the first. Instead of adding the new element when the Add button is pressed, that action is performed when the user presses the Enter key. To test for this event, we can write a method for the combo box's `OnKeyPress` event and check whether the key is the Enter key , which has the numeric code 13. The remaining statements are similar to those of the button's `OnClick` event:

```
procedure TForm1.ComboBox3KeyPress(
  Sender: TObject; var Key: Char);
begin
  {if the user presses the Enter key}
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

Delphi's `DateTimePicker` component has a user interface similar to that of a ComboBox.

Choosing a Value in a Range

The last basic component I want to explore in this chapter is the scroll bar. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. Notice, however, that when a scroll bar is associated with another component, it is really a portion of that component — one of its properties — and there is little relationship to the ScrollBar component itself. Forms having a scroll bar have no ScrollBar component. A portion of their border is used to display that graphical element.

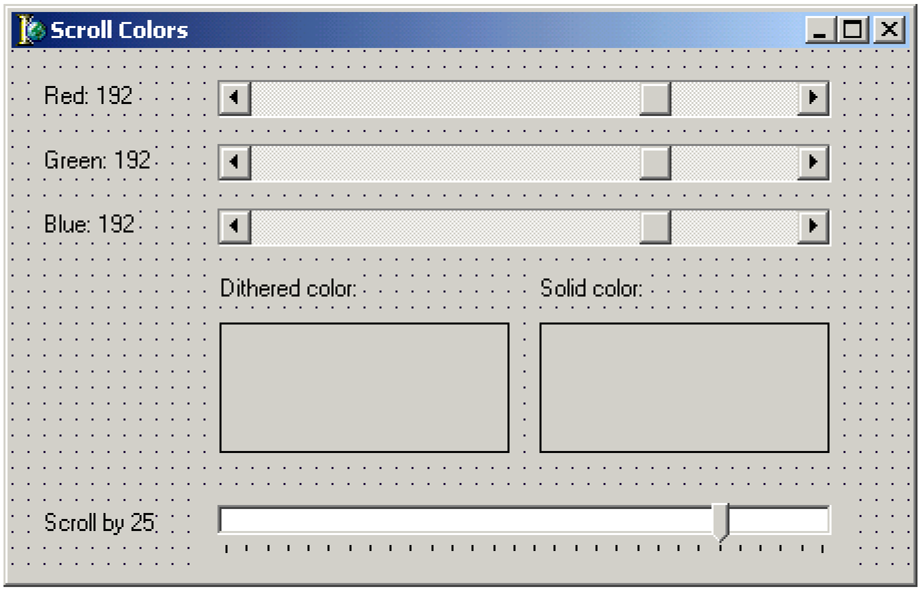
Direct usage of the ScrollBar component is quite rare, especially with the TrackBar Windows common control. However, there are cases in which it can play a role. The typical example is to allow a user to choose a numerical value in a large range (since a TrackBar is generally used for smaller ranges).

Most Windows programming books describe scroll bars using the example of selecting a color, and this book is no exception. But if you've seen a typical Windows example, you'll notice something very interesting: using Delphi, you can build this example in about one-fourth the time and writing a minimal amount of code.

The Scroll Color Example

The ScrollC example — the name stands for scroll color — has a simple form with three scroll bars and three corresponding labels, a track bar with its own label, and some shape components to show the current color. Each scroll bar refers to one of the three fundamental colors, which in Windows are red, green, and blue (RGB). Each label displays the name of the corresponding color and the current value.

Scroll bars have a number of peculiar properties. You can use `Min` and `Max` to determine the range of possible values; `Position` holds the current position; and the `LargeChange` and `SmallChange` properties indicate the increment caused by clicking on the bar or on the arrow at the end of the bar, respectively.



In the ScrollC example, the value of each bar ranges from 0 to 255. The range is determined by the fact that each color is a DWORD with the lower three bytes representing the Red, Green, and Blue values (which is how colors are represented in Windows and in the VCL). The initial value of 192 has been chosen for the position because with settings of 192 for red, 192 for green, and 192 for blue, you get the typical light gray, which is the default value for the color of the form and of the shapes. Here is the textual description of one of these three ScrollBar components:

```
object ScrollBarRed: TScrollBar
  LargeChange = 25
  Max = 255
  Position = 192
  OnScroll = ScrollBarRedScroll
end
```

The TrackBar components has similar properties (Min, Max, Position):

```
object TrackBar1: TTrackBar
  Max = 30
  Min = 1
  Orientation = trHorizontal
  Frequency = 1
  Position = 25
  TickMarks = tmBottomRight
  TickStyle = tsAuto
  OnChange = TrackBar1Change
end
```

This control is used, in this example, to set the LargeChange property of the three scrollbars, with the following code:

```
procedure TFormScroll.TrackBar1Change(Sender: TObject);
begin
  LabelScroll.Caption := 'Scroll by ' +
    IntToStr(TrackBar1.Position);
  ScrollBarGreen.LargeChange := TrackBar1.Position;
  ScrollBarRed.LargeChange := TrackBar1.Position;
  ScrollBarBlue.LargeChange := TrackBar1.Position;
end;
```

When one of the scroll bars changes (the OnScroll event), the program has to update the corresponding label and the color of the shapes. The first of these shapes is used to show the color as it is determined by the three RGB values of the scroll bars. Assigning the color to the Color property of the brush used to fill the surface of the shape, we obtain a dithered color, an approximation of the real tint made with the colors available on the video adapter. The same color is assigned to the Color property of the pen of the second shape, resulting in the closest approximation of the requested color. Pens, in fact, do not use dithering, but rather the closest *pure* color. You can see the difference by running this example, although the effect might change depending on your video adapter.

If you browse through the code of the program, notice also that there is a third shape component used to mimic the border of the second shape. The real border of this shape, in fact, is enlarged to fill its whole surface, using a very wide pen. This way we use the color of the pen — the wide border — to actually fill the shape. Here is the code corresponding to one of the scroll bars:

```
procedure TFormScroll.ScrollBarRedScroll(Sender: TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  LabelRed.Caption := 'Red: ' + IntToStr(ScrollPos);
  Shape1.Brush.Color := RGB (ScrollBarRed.Position,
    ScrollBarGreen.Position, ScrollBarBlue.Position);
  Shape2.Pen.Color := RGB (ScrollBarRed.Position,
    ScrollBarGreen.Position, ScrollBarBlue.Position);
end;
```

You need to copy this code once for each scroll bar and correct the name of the label and its output text. The second and third statements always remain the same. They are based on a Windows function, RGB, which takes three values in the range 0–255 and creates a 32-bit value with the code of the corresponding color.

It is interesting to note that the OnScroll event has three parameters: the sender, the kind of event (ScrollCode), and the final position of the thumb (ScrollPos). This type of event can be used for very precise control of the user's actions. The ScrollCode parameter indicates if the user is dragging the thumb (scTrack, scPosition, or scEndScroll), has clicked on one of the two final arrows (scLineUp or scLineDown), has clicked on the bar in one of the two directions (scPageUp or scPageDown), or is trying to scroll out of the range (scTop or scBottom).

What's Next

In this chapter, we have started to explore some of the basic components available in Delphi. These components correspond to the standard Windows controls and some of the Windows common controls, and are extremely common in applications (with the exception of the stand-alone scroll bars). Of course, when you start adding more advanced Delphi components to an application, you can easily build more complex and colorful user interfaces and more powerful programs.

The next chapter is devoted to a specific and important topics: the use of menus. After that, we'll see more about forms, have a light excursus on multimedia, and wrap up the book with some simple specific techniques.

Chapter 5: Creating and Handling Menus

- The structure of a menu
- Using menu templates
- Checking, disabling, and modifying menus at run-time
- Creating menu items at run-time
- A custom menu check mark
- Bitmap menu items and owner-draw menu items
- The system menu
- Pop-up menus

The sample programs we have built so far have lacked one of the most important user-interface elements of any Windows application: the menu bar. Although our forms have each had a system menu, its use has been very limited. In practical applications, however, the menu bar is a central element in the development of a program. While the user can click and sometimes drag the mouse to select options, most complex tasks usually involve menu commands. Consider the applications you use and the number of menu commands you issue in those programs (including those invoked by a shortcut key, such as Ctrl+C, which is equivalent to the Edit | Copy command in most applications).

Menus are so important that almost any real Windows application has at least one. In fact, an application can also have several menus that change at run-time (more on this later), various local menus (usually activated with a right mouse click), and even a customized system menu.

The Borland programmers who created Delphi considered menus so important that they have placed the corresponding components in the Standard page of the Components palette.

The Structure of the Main Menu

Before looking at the use of menus in Delphi, let me recap some general information about menus and their structure. Usually, a menu has two levels. A menu bar, appearing below the title of the window, contains the names of the pull-down menus, each of which in turn contains a number of items. However, the menu structure is very flexible. It is possible to place a menu item directly in the menu bar and to place a *second level* pull-down menu inside another pull-down menu.

You should avoid placing commands directly on the menu bar, because users tend to select the elements of the menu bar to explore the structure of the menu. They do not expect to issue a command this way. If, for some reason, you really need to place a command in the menu bar, at least place the standard exclamation mark after it. Using an exclamation mark is a standard hint, but most users have never seen this “convention,” so it’s best to avoid the whole situation altogether and simply have a pull-down menu with a single menu item. A typical example is a Help menu with a single About menu item.

Putting a pull-down menu inside another pull-down menu — a second-level pull-down — is far more common, and Windows in this case provides a default visual clue, a small triangular glyph at the right of the menu. Many applications use this technique, because the system makes heavy use of multilevel menus (consider the Programs menu of the Start button). However, keep in mind that selecting a menu item in a second-level pull-down takes more time and can become tedious.

Many times, instead of having a second-level pull-down, you can simply group a number of options in the original pull-down and place two separator bars, one before and one after the group. You can see an exaggerated multilevel menu in the Levels example in the source code. Since this is a demonstration of what you should try to *avoid*, I won't list the structure of the menu here.

Different Roles of Menu Items

Now let's turn our attention to menu items, regardless of their position in the menu structure. There are three fundamental kinds of menu items:

- *Commands* are menu items used to execute an action. They have no special visual clue.
- *State-setters* are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active. In this case, selecting the command produces the opposite action.
- *Dialog menu items* are menu items that cause a dialog box to appear. The real difference between these and the other menu items is that a user should be able to explore the possible effects of the corresponding dialog box and eventually abort it by choosing the Cancel button. These commands should have a visual clue, consisting of an ellipsis (three dots) after the text.

> Besides the traditional state-setters with a check mark, you can also have radio menu items with a bullet check mark. These menu items represent alternative selections, just as RadioButton components do, and simply checking one of them disables the other elements of the group. We'll explore radio menu items in an example later on.

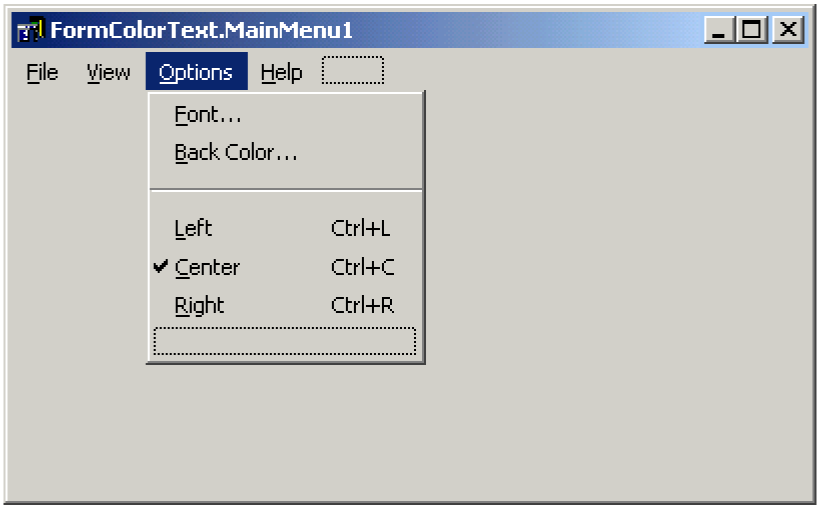
Building a Menu with the Menu Designer

Delphi includes a special editor for menus, the Menu Designer. To invoke this tool, place a MainMenu component on a form and double-click on it. Don't worry too much about the position of the menu component on the form, since it doesn't affect the result; the menu is always placed properly, below the form's caption.

> To be more precise, the form displays, below its caption, the menu indicated in its `Menu` property, which is set by default as soon as you create the first main menu component of the form. If the form has more than one main menu component, this property should be set manually and can be changed both at design-time and at run-time.

The Menu Designer is really powerful: It allows you to create a menu simply by writing the text of the commands, to move the items or pull-down menus by dragging them around, and to set the properties of the items easily. It is also very flexible, allowing you to place a command directly in the menu bar (this happens each time you do not write any element in the corresponding pull-down menu) or to create second-level pull-down menus.

To accomplish this, select the Create Submenu command on the Menu Designer's local menu (the local menu invoked with the right mouse button).



Another very important feature available through the Menu Designer is the ability to create a menu from a template. You can easily define new templates of your own. Simply create a menu, and use the Save As Template command on the local menu to add it to the list. This makes sense particularly if you need to have a similar menu in two applications or in two different forms of the same application.

The Standard Structure of a Menu

If you've used Windows applications for some time, you have certainly noticed that the structure of an application's menu is not an invention of its programmers. There are a number of standard Windows guidelines describing how to arrange the commands in a menu. You can infer most of these rules by looking at the menus of some of the best-selling applications.

An application's menu bar should start with a File pull-down, followed by Edit, View, and then some commands specific to the application. The final part of the sequence includes Options, Tools, and Window (in MDI, or Multiple Document Interface, applications) and always terminates with Help. Each of these pull-down menus has a standard layout, although the actual items depend on the application. The File menu, for example, usually has commands such as New, Open, Save, Save As, Print, Print Setup, and Exit.

Shortcut Keys and Hotkeys

A common feature of menu items is that they contain an underlined letter, generally called a *hotkey*. This letter, which is often the first letter of the text, can be used to select the menu using the keyboard. Pressing Alt plus the underlined key selects the corresponding pull-down menu. By pressing another underlined key on that menu, you issue a command.

Of course, each element of the menu bar must have a different underlined character. The same is true for the menu items on a specific pull-down menu. (Obviously, menu items on different pull-down menus can have the same underlined letter.) To indicate the underlined key, you simply place an ampersand (&) before it, as in Save &As... or &File. In these examples, the underlined keys would be A for Save As and F for File.

Menu items have another standard feature: shortcut keys. When you see the shorthand description of a key, or key combination, beside a menu item, it means you can press those keys to give that command. Although giving menu commands with the mouse is easier, it tends to be somewhat slow, particularly for keyboard-intensive applications, since you have to move one of your hands from the keyboard to the mouse. Pressing Alt and the underlined letter might be faster, but it still requires two operations. Using a shortcut key usually involves pressing a special key and another key at the same time (such as Ctrl+C). Windows doesn't even display the corresponding pull-down menu, so this results in a faster internal operation, too.

In Delphi, associating a shortcut key with a menu item (pull-down menus cannot have a shortcut key) is very easy. You simply select a value for the ShortCut property, choosing one of the standard combinations: Ctrl or Shift plus almost any key.

You might even add shortcut keys to a program without adding a real menu. For example, you can create a pop-up menu, connect it to a form (by setting the PopupMenu property of the form), set the Visible property of all of its items to False, and add the proper shortcut keys; a user will never see the menu, but the shortcuts (documented in your Help system, of course) will work. If this is not clear, you can look at the HShort example (the name stands for "Hidden Shortcut") in the book code.

Using the Predefined Menu Templates

To let you start developing an application's menu following the standard guidelines, Delphi contains some predefined menu templates. The templates include two different File pull-down menus, an Edit menu (including OLE commands), a Window menu, and two Help menus. There is also a complete MDI menu bar template, which has the same four menu categories.

Using these standard templates brings you some advantages. First of all, it is faster to reuse an existing menu than to build one from scratch. Second, the menu template follows the standard Windows guidelines for naming menu commands, for using the proper shortcuts, and so on. Of course, using these menus makes sense in a file-based application. But if the program you are writing doesn't handle files, has no editing capabilities, and is not MDI, you'll end up using only the template Help pull-down menu.

Responding to Menu Commands

To build the MenuOne example, the first example with a menu, we will extend the LabelCo example of the last chapter. The new version of the form has been extended with a MainMenu component. This menu bar has four pull-down menus: the File pull-down with only the Exit option, the View pull-down with only the Toolbar menu item, the Options menu with various options, and the Help menu with the About menu item.

To add the separator in the Options pull-down menu, simply insert a hyphen as the text of the command. Do not change the Break property. (Except for rare situations, the Break property will make a mess of your menu. You are better off forgetting that this property even exists.)

> Of course, the `Break` property has its uses, or it would not have been added to the component. You can better understand it if I use different names for its possible values, NewLine or NewColumn. If an item on the menu bar has the `mbMenuBarBreak` (or NewLine) value, this item will be displayed in a second or subsequent line. If a menu item has the `mbMenuBreak` (or NewColumn) value, this item will be added to a second or subsequent column of the pull-down. Neither of these features are used very often.

The Code Generated by the Menu Designer

Once you have built this menu, take a look at the list of components displayed by the Object Inspector, or open the `DFM` file with the textual description of the form, which will also contain a textual description of the menu structure. Here is the portion of the textual description of the form related to the menu and its items:

```
object MainMenu1: TMainMenu
  object File1: TMenuItem
    Caption = '&File'
    object Exit1: TMenuItem
      Caption = 'E&xit'
      OnClick = Exit1Click
    end
  end
  object View1: TMenuItem
    Caption = '&View'
    object Toolbar1: TMenuItem
      Caption = '&Toolbar'
      Checked = True
      OnClick = Toolbar1Click
    end
  end
  object Options1: TMenuItem
    Caption = '&Options'
    object Font1: TMenuItem
      Caption = '&Font...'
      OnClick = Font1Click
    end
    object BackColor1: TMenuItem
      Caption = '&Back Color...'
      OnClick = BackColor1Click
    end
    object N1: TMenuItem
      Caption = '-'
    end
    object Left1: TMenuItem
      Caption = '&Left'
      ShortCut = 16460 // stands for Ctrl+L
      OnClick = Left1Click
    end
    object Center1: TMenuItem
      Caption = '&Center'
      Checked = True
      ShortCut = 16451 // stands for Ctrl+C
```

```
        OnClick = Center1Click
      end
      object Right1: TMenuItem
        Caption = '&Right'
        ShortCut = 16466 // stands for Ctrl+R
        OnClick = Right1Click
      end
    end
    object Help1: TMenuItem
      Caption = '&Help'
      object About1: TMenuItem
        Caption = '&About Menu One...'
        OnClick = About1Click
      end
    end
  end
```

As you can see in the listing above, there is a specific component for each menu item, one for each pull-down menu, and, surprisingly, even one for each separator. Delphi builds the names of these components automatically when you insert the menu item's label. The rules are simple:

- Any blank or special character (including ampersands and hyphens) is removed.
- If there are no characters left, the letter `N` is added.
- A number is always added at the end of the name (1 if this is the first menu item with this name, a higher number if not).

All of these new components are listed in the Object Inspector, and you can select them directly or navigate among them by opening the Menu Designer and selecting menu items visually. Actually each of these items is also listed as a component in the class definition of the form:

```
type
  TFormColorText = class(TForm)
    MainMenu1: TMainMenu;
    Options1: TMenuItem;
    Font1: TMenuItem;
    BackColor1: TMenuItem;
    N1: TMenuItem;
    Left1: TMenuItem;
    Center1: TMenuItem;
    Right1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    File1: TMenuItem;
    Exit1: TMenuItem;
    View1: TMenuItem;
    Toolbar1: TMenuItem;
    ...
```

> If there are menu items your code will not refer to (such as the separators), you can actually delete the fields declaring these objects (as `N1` above). They will be created anyway, since they are listed in the form definition file, but you won't be able to access them easily from within the source code of the form. Since the objects are created anyway, you won't save much memory, though (only the space for the reference inside the form class), but removing useless statements may improve code readability.

To respond to menu commands, you should define a method for the `OnClick` event of each menu item. The `OnClick` event of the pull-down menus is used only in special cases — for example, to check whether the menu items below should be disabled. The `OnClick` event of the separators is totally useless, because it will never be activated.

> Once you have defined the main menu of a form and it is displayed below the caption, you can add a new method for the `OnClick` event of a menu command simply by selecting it in the menu bar. If a handler is already present, Delphi will show you the corresponding portion of the source code; otherwise, a new method will be added to the form.

The Code of the MenuOne Example

The code of the MenuOne example is very simple, and is similar to that of the LabelCo example it extends. The `OnClick` event of the menu command for the background color and of the corresponding toolbar button are connected to the same method:

```
object BtnBackColor: TButton
  Caption = '&Back Color...'
  OnClick = BackColor1Click
end
```

This method has the same code as the earlier version. The menu command and button related to the font are both connected to the `Font1Click` method, which this time displays the font selection dialog box, not the color selection dialog box. Notice that the code is quite compact, because it uses a `with` statement:

```
procedure TFormColorText.Font1Click(Sender: TObject);
begin
  with FontDialog1 do
  begin
    Font := Label1.Font;
    if Execute then
      Label1.Font := Font;
  end;
end;
```

The other three menu items of the Options pull-down menu (and the last three buttons of the toolbar) have basically the same code as the LabelCo example: the code of their `OnClick` event handlers simply set the `Alignment` property of the label. This works fine, but the resulting application doesn't follow the standard user-interface guidelines. Each time you have a series of choices in a menu, the selected choice should have a check mark beside it.

To accomplish this, you need to create two different operations. First, you have to place a check mark near the default choice, Center, changing the value of the menu item's `Check` property in the Object Inspector. Second, you should correct the code so that each time the selection changes, the check mark is properly set:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Center1.Checked := False;
  Right1.Checked := False;
end;
```

The other two methods are similar. You can simply copy the source code of the last three statements, paste this text twice into the other two methods, and correct the values of the three `Checked` properties so that each time one of them is set to `True`. Removing the check marks from the other items of the group can be handled in more efficient ways, but the real solution is to use radio menu items, instead of check marks. We'll look at this technique later on.

The View | Toolbar menu item is a typical item with a check mark, set when the toolbar is visible. Here is its code:

```
procedure TFormColorText.Toolbar1Click(Sender: TObject);
begin
  Panel1.Visible := not Panel1.Visible;
  Toolbar1.Checked := Panel1.Visible;
end;
```

The program toggles the status of the `Visible` property of the panel, then sets the check mark — the `Checked` property — of the menu item accordingly. You should only remember to set the initial value of this property, to match the initial status of the panel. The Help | About and File | Exit commands simply show a message box or call the `Close` method of the form, respectively. They are so simple I won't show their source code here. (You can find it in the `MenuOneF.PAS` file among th source code.)

Modifying the Menu at Run-Time

You can perform a number of operations in Windows to change the structure of a menu at run-time. We'll start by examining in detail the operations you can do on a single menu item, then move to pull-down menus, build a flexible main menu, and change the default bitmap for the check mark.

Changing Menu Items at Run-Time

It is important to note that menu items can change at run-time. For example, when a menu command cannot or should not be selected, it is usually grayed. In this case, the user has no way to issue that command. You shouldn't generally hide a menu item when it is not available, but simply disable it. This standard technique lets users know that the command is currently not available. Otherwise, they might think the menu command was somewhere else in the menu structure, and keep looking for it.

Another visual change is the use of the check mark, which applications can toggle on and off easily, as we've seen in the last example. At times, to implement a state-setter menu item, you can change the text of the menu item altogether, which might result in an easier interface. For example, suppose an application has a Show Toolbar command. If you select it, a toolbar will appear and a check mark will be added to the item. This means that if you again select the Show Toolbar command, the toolbar will disappear: The command you issue has the opposite effect as its name. To avoid this problem, you might use two different captions for the two states of the menu item, such as *Show Toolbar* and *Hide Toolbar*.

When the user can select more than two choices, it is better to use multiple menu items with a check mark, or even better the new radio menu item user interface, which is becoming the standard approach.

Three properties are commonly used to modify a menu item:

- We used the `Checked` property in the example above to add or remove a check mark beside the menu item.
- The `Enabled` property can be used to gray a menu item so that it cannot be selected by a user (but it remains visible).
- The last property of this group is the `Caption`, the text of the menu item, which can be modified to reflect the actual effect of a command, as discussed above.

I'll demonstrate the use of these properties by extending the MenuOne example (the name of the new project is MenuOne2) with new menu items. I've added two new menu items to the View pull-down menu (Hide Label and Fixed Font) and two new menu items in the Options pull-down menu (Fixed View and Disable Help), plus a couple of new separators.

The Hide Label menu item demonstrates the use of different captions for an item depending on the status of the program. Here is its `OnClick` event handler:

```
procedure TFormColorText.HideLabel1Click(Sender: TObject);
begin
  Label1.Visible := not Label1.Visible;
  if Label1.Visible then
    HideLabel1.Caption := 'Hide &Label'
  else
    HideLabel1.Caption := 'Show &Label'
end;
```

Disabling Menu Items and Hiding Pull-Down Menus

The other three new menu items of the MenuOne2 example are used to disable or hide other menu items or pull-down menus. The View | Fixed Font command is used to disable the Options | Font menu item:

```
procedure TFormColorText.FixedFont1Click(Sender: TObject);
begin
  ToggleCheck (FixedFont1);
  Font1.Enabled := not Font1.Enabled;
end;
```

This method calls the custom `ToggleCheck` procedure I've written as a shortcut to the code for toggling the check mark (something all these three final menu commands do). Here is its simple code:

```
procedure ToggleCheck (Item: TMenuItem);
begin
  Item.Checked := not Item.Checked;
end;
```

I've already advised you against hiding menu items (by turning off their `Visible` property), because users will probably try to find them in a different pull-down menu. Menu items should generally be disabled when you want to prevent users from calling them. Hiding entire pull-down menus, however, is a common practice. In many applications the pull-down menus you see reflect the window you are working on. Technically, it is possible to disable a pull-down menu as well, but this is not very common. The last two menu items I've added to the program do these two operations, as you can see in their corresponding `OnClick` event handlers:

```
procedure TFormColorText.FixedView1Click(Sender: TObject);
begin
  ToggleCheck (FixedView1);
  View1.Visible := not View1.Visible;
end;

procedure TFormColorText.DisableHelp1Click(Sender: TObject);
begin
  ToggleCheck (DisableHelp1);
  Help1.Enabled := not Help1.Enabled;
end;
```

The View pull-down menu has been removed, and the Help pull-down menu is grayed.

Using Radio Menu Items

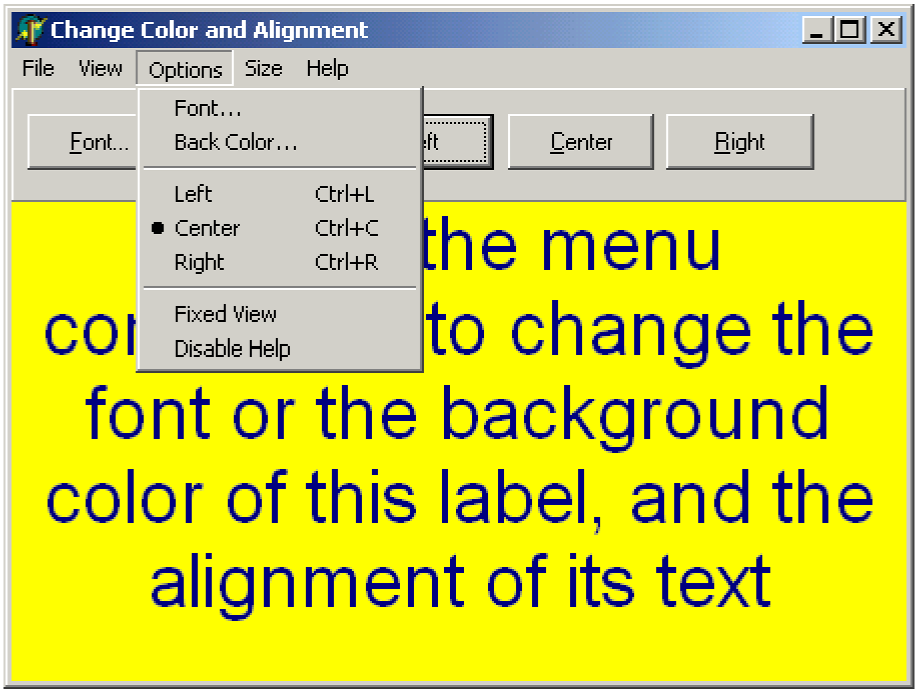
In addition to using check marks, in Windows you can use radio menu items. These provide not only a different user interface, but also different behavior (basically simpler code, since the system does some of the work for us). A notable example of this new user-interface feature is the View menu of the Windows Explorer.

In Delphi, simply set the `RadioItem` property of a MenuItem component to `True` and you get the new check mark for the item. If you set this property for several consecutive menu items and set their `GroupIndex` property to the same value, they'll use the new mark and behave as radio buttons. This means that only one of the menu items in the group will be selected at a time. Instead of having to deselect all other items manually, as you did in the first version of the MenuOne example, now you can simply select the proper menu item, and the rest is automatic.

Here is how you can implement this feature, in the third version of the MenuOne example (which also has another feature I'll discuss in the next section). The following listing shows the updated textual description of this group of menu items:

```
object Options1: TMenuItem
  Caption = '&Options'
  ...
  object Left1: TMenuItem
    Caption = '&Left'
    GroupIndex = 1
    RadioItem = True
    OnClick = Left1Click
  end
  object Center1: TMenuItem
    Caption = '&Center'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = Center1Click
  end
```

```
object Right1: TMenuItem
  Caption = '&Right'
  GroupIndex = 1
  RadioItem = True
  OnClick = Right1Click
end
```



This code will work as is, but we can actually simplify it. Here is the new version of the `Right1Click` method:

```
procedure TFormColorText.Right1Click(Sender: TObject);
begin
  Label1.Alignment := taRightJustify;
  // Left1.Checked := False; // now useless
  // Center1.Checked := False; // now useless
  Right1.Checked := True;
end;
```

I've simply commented out the two useless statements, instead of deleting them, to let you see the differences from the older version.

Creating Menu Items Dynamically

The run-time changes on menu items and pull-down menus we've seen so far were all based on the direct manipulation of some properties. These components, however, also have some interesting methods, such as `Insert` and `Remove`, that you can use to make further changes.

The basic idea is that each object of the `TMenuItem` class — which Delphi uses for both menu items and pull-down menus — contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each submenu has a list of submenus, each with its own list of submenus, and so on.

The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`, which contains the number of subitems. Adding new menu items (or entire pull-down menus) to a menu is fairly easy. Slightly more complex is the handling of the commands related to the new menu items. Basically, you need to write a specific message-response method in your code (without any help from the Delphi environment), and then assign it to the new menu item by setting its `OnClick` property. As an alternative, you can have a single method used for several `OnClick` events and use its `Sender` parameter to determine which menu command the user issued.

All these features are demonstrated by the MenuOne3 example. As soon as you start this program, it creates a new pull-down with menu items used to change the size of the font of the big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you. I could have created the pull-down at design-time, and then added the menu items dynamically, but I prefer showing you the complete code, so that you can apply it to other cases.

To create a new menu item (or pull-down) you simply call the `Create` constructor of the `TMenuItem` class:

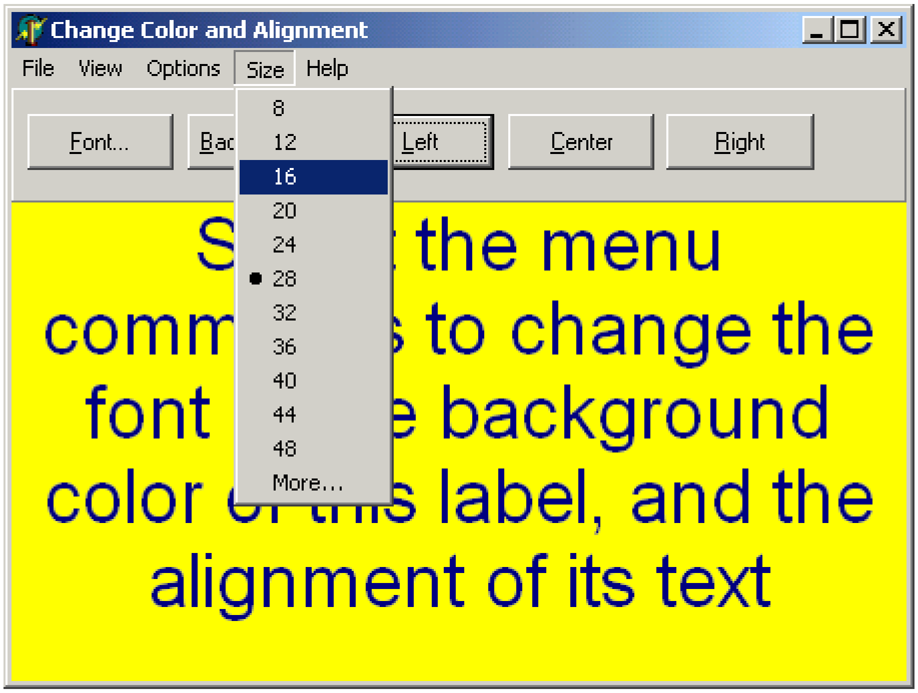
```
var
  PullDown: TMenuItem;
begin
  PullDown := TMenuItem.Create (self);
```

Then you can simply set its `Caption` and other properties, and finally insert it in the proper parent menu. The new pull-down should be inserted in `Items` of the `MainMenu1` component. You can calculate the position, knowing the index is zero-based, or you can ask the main menu component for the previous pull-down menu:

```
Position := MainMenu1.Items.IndexOf (Options1);
MainMenu1.Items.Insert (Position + 1, PullDown);
```

The menu items of this pull-down are created in a `while` loop (I don't use a `for` loop because I want to provide only one menu item for every four possible sizes: 8, 12, 16, and so on). The code to create each item is slightly more complex simply because I want to turn them into radio items, but the basic structure of the code is very simple:

```
var
  Item: TMenuItem;
  I: Integer;
begin
  ...
  I := 8;
  while I <= 48 do
  begin
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
```


Change Color and Alignment
File
View
Options
Size
Help
Font...
Bac
Left
Center
Right
8
12
16
20
24
● 28
32
36
40
44
48
More...
S the menu comm to change the font background color label, and the alignment of its text

To insert an item at the end I call the Insert method passing the number of items (`PullDown.Count`) as a parameter. As you can see, the program adds one extra item at the end of the menu, used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method, which shows the font selection dialog box:

```
Item := TMenuItem.Create (self);
Item.Caption := 'More...';
Item.OnClick := Font1Click;
PullDown.Insert (PullDown.Count, Item);
```

Also the `OnClick` events of the other menu items are connected with a method, but this time it is a method you have to define manually, by adding its declaration to the form class:

```
type
  TFormColorText = class(TForm)
    ...
  public
    procedure SizeItemClick(Sender: TObject);
end;
```

The method should have the proper signature (or parameters list). Here is the code of the method, which is based on the `Sender` parameter:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
end;
```

As you can see, this code doesn't set the proper check mark (or radio item mark) next to the selected item. The reason is that the user can select a new size by changing the font. For this reason, we can use a different approach, and handle the `OnClick` event of the pull-down menu. This event is activated just before showing the pull-down menu, so we can use it to set the proper check mark at that time:

```
procedure TFormColorText.SizeClick (Sender: TObject);
```

```
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) =
        Label1.Font.Size then
      begin
        Items [I].Checked := True;
        Found := True;
        System.Break; // skip the rest of the loop
      end;
    if not Found then
      Items [Count - 1].Checked := True;
  end;
end;
```

This code scans the items of the pull-down menu we have activated (the `Sender`), skipping only the final one, and checks whether the caption matches the current `Size` of the font of the label. If no match is found, the program checks the last menu item, to indicate that a different size is active.

Of course we have to set this `SizeClick` event handler at run-time, when the pull-down menu is created. So we can finally look at the complete source code of the `FormCreate` method, which sums up all the features I have discussed in this section:

```
procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pulldown menu
  PullDown := TMenuItem.Create (self);
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
  Position := MainMenu1.Items.IndexOf (Options1);
  MainMenu1.Items.Insert (Position + 1, PullDown);

  // create menu items for various sizes
  I := 8;
  while I <= 48 do
  begin
    // create the new item
    Item := TMenuItem.Create (self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;

  // add extra item at the end
  Item := TMenuItem.Create (self);
  Item.Caption := 'More...';
  // make it a radio item
  Item.GroupIndex := 1;
  Item.RadioItem := True;
  // handle click by showing the font dialog box
  Item.OnClick := Font1Click;
  PullDown.Insert (PullDown.Count, Item);
end;
```

Creating Menus and Menu Items Dynamically

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the MenuOne3 example. As an alternative, you can also use some global functions available in the Menus unit:

```
function NewMenu(Owner: TComponent; const AName: string;
  Items: array of TMenuItem): TMainMenu;
function NewPopupMenu(Owner: TComponent;
  const AName: string; Alignment: TPopupAlignment;
  AutoPopup: Boolean; Items: array of TMenuitem):
  TPopupMenu;
function NewSubMenu(const ACaption: string;
  hCtx: Word; const AName: string;
  Items: array of TMenuItem): TMenuItem;
function NewItem(const ACaption: string;
  AShortCut: TShortCut; AChecked, AEnabled: Boolean;
  AOnClick: TNotifyEvent; hCtx: Word;
  const AName: string): TMenuItem;
function NewLine: TMenuItem;
```

The `NewMenu` and `NewPopupMenu` functions, in particular, should be used to create brand-new menus. Calling the constructors of the corresponding classes, in fact, doesn't always work properly.

Short and Long Menus

If you don't like creating menu items dynamically, but still need to have a very flexible menu, there are a couple of good alternatives. You can create a large menu with all the items you need, then hide all the items and pull-down menus you do not want at the beginning. To add a new command you need only show it. This solution is a follow-up to what we have done up to now.

You can also create several menus, possibly with common elements, and exchange them as required. This approach is demonstrated in this section. A typical example of a form having two menus is one that uses two different sets of menus (long and short) for two different kinds of users (expert and inexperienced). This technique was common in major Windows applications for some years but has since been replaced by other approaches, such as letting each user redefine the whole structure of the menu.

The idea is simple and its implementation straightforward:

1. Prepare the full menu of the application, adding a menu item with the `Caption` *'Short'*.
2. Add this menu to the Delphi menu template.
3. Place a second MainMenu component on the form, and copy its structure from the template.
4. In the second menu, remove the items corresponding to advanced features and change the `Caption` of the special item from *'Short'* to *'Long'*.
5. In the Menu property of the form, set the MainMenu component you want to use when the application starts, choosing one of the two available. Note that this operation has an effect on the form at design-time, too.
6. Write the code for the Short and Long commands so that when they are selected, the menu changes.

If you follow these steps, you'll end up with an application similar to TwoMenus, which can change its menu at run-time. The example has two different MainMenu components, with useless "dummy" menu items, plus the Short and Long commands.

The application does nothing apart from changing the main menu when the Short Menu or Long Menu items are selected. Here is the code for the Short Menu item:

```
procedure TForm1.ShortMenus1Click(Sender: TObject);
begin
  {activate short menu}
  Form1.Menu := MainMenu2;
end;
```

Graphical Menu Items

Besides the run-time changes on a menu's structure I've listed so far, which are all directly available in Delphi, there are a number of operations you can perform on menus using the Windows API. In fact, there are several API functions referring to menus.

In particular, using Windows API functions for menus lets us add some graphics to them. We can customize the check mark, replace the strings with bitmaps, and even paint in the menu items.

Customizing the Menu Check Mark

As I've just mentioned, there are a number of ways to customize a menu in Windows. In this section, I'm going to show you how you can customize the check mark used by a menu item, using two bitmaps of your own. This example, NewCheck, involves using bitmaps and calling a Windows API function.

First I should explain why we need two bitmaps, not just one. If you look at a menu item, it can have either a check mark or nothing. In general, however, Windows uses two different bitmaps for the checked and unchecked menu item. I've prepared two bitmaps, of 16 x 14 pixels, using the Delphi Image Editor. You can easily run this program from the Tools menu, but you can prepare the bitmaps with any editor, including Windows Paintbrush. The bitmaps should be stored in two BMP files in the same directory as the project.

The NewCheck example has a very simple form, with just two components, a MainMenu and a label:

```
object Form1: TForm1
```

```
  Caption = 'New Check'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  object Label1: TLabel
    Alignment = taCenter
    AutoSize = False
    Caption = 'OFF'
    Font.Height = -96
    Font.Name = 'Arial'
    Font.Style = [fsBold]
  end
  object MainMenu1: TMainMenu
    object Command1: TMenuItem
      Caption = '&Command'
      OnClick = Command1Click
      object Toggle1: TMenuItem
        Caption = '&Toggle'
        OnClick = Toggle1Click
      end
    end
  end
end
```

As you can see in the listing above, the menu item has a single command (Toggle), which will be used to change the text of the label from *'ON'* to *'OFF'* and change the check mark, too:

```
procedure TForm1.Toggle1Click(Sender: TObject);
begin
  Toggle1.Checked := not Toggle1.Checked;
  if Toggle1.Checked then
    Label1.Caption := 'ON'
  else
    Label1.Caption := 'OFF';
end;
```

The most important portion of the code of this example is the call to the `SetMenuItemBitmaps` Windows API function:

```
function SetMenuItemBitmaps (Menu: HMenu;
  Position, Flags: Word;
  BitmapUnchecked, BitmapChecked: HBitmap): Bool;
```

This function has a number of parameters:

- The first parameter is the pull-down menu we refer to.
- The second parameter is the position of the menu item in that pull-down menu.
- The third parameter is a flag that determines how to interpret the previous parameter (Position).
- The last two parameters indicate the bitmaps that should be used.

Notice that this function changes the check mark bitmaps only for a specific menu item. Here is the code you can use in Delphi to call the function:

```
procedure TForm1.Command1Click(Sender: TObject);
begin
  SetMenuItemBitmaps (Command1.Handle,
    Toggle1.Command, MF_BYCOMMAND,
    Bmp2.Handle, Bmp1.Handle);
end;
```

This call uses two bitmap variables that are defined in the code and the names of some components (`Command1` is the name of the pull-down, and `Toggle1` is the name of the menu item). The code above shows that it is usually very easy to pass the handle of an element to a Windows function — just use its `Handle` property.

At first I thought this function could be called when the form was created, after the two bitmaps had been loaded from the file, but it cannot. Delphi changes the default Windows behavior somewhat, forcing the application to re-associate the bitmap with the menu items each time they are displayed. The solution I've found is to execute this call each time the pull-down menu is selected — that is, on the `OnClick` event of the pull-down.

The only thing left is to load the bitmaps. You need to add two fields of the `TBitmap` type to the form class, create an instance of the two objects, and then load the bitmaps from the two `BMP` files. This is done only once, when the form is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Bmp1 := TBitmap.Create;
  Bmp2 := TBitmap.Create;
  Bmp1.LoadFromFile ('ok.bmp');
  Bmp2.LoadFromFile ('no.bmp');
end;
```

The two bitmaps should also be destroyed when the program terminates (in the handler of the `OnDestroy` event of the form):

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Bmp1.Free;
  Bmp2.Free;
end;
```

Notice that to run this program, you need to have the two `BMP` files in the same directory as the executable file. The bitmaps, in fact, are loaded at run-time and are not embedded by Delphi in the `EXE` file. As an alternative I could have used two non-visible Image components to hold the images.

> You can indeed include a bitmap in the resources of an application and in its executable file in order to be able to ship the application in a single file. This process, however, is slightly more complex, so I've decided not to use it here.

Bitmap Menu Items

Instead of placing a bitmap close to a menu item to indicate the status of its `Checked` property, as we've done in the previous section, you can actually replace the text of a menu item with a bitmap. In specific cases this can make an application easier to use.

BitMenu is a very simple program. I've put a shape in the middle of a form and added a menu to set the kind of shape (rectangle, rounded rectangle, or ellipse), and its color. Here is the textual description of the form and its menu:

```
object Form1: TForm1
  Caption = ' Bitmap Menu'
  Menu = MainMenu1
  OnCreate = FormCreate
  OnDestroy = FormDestroy
  OnResize = FormResize
  object ShapeDemo: TShape...  // default properties
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Exit1: TMenuItem...
    end
    object Shape1: TMenuItem
      Caption = '&Shape'
      object Ellipse1: TMenuItem
        Caption = 'Ellipse'
        OnClick = Ellipse1Click
      end
      object RoundRec1: TMenuItem
        Caption = 'RoundRec'
        OnClick = RoundRec1Click
      end
      object Rectang1: TMenuItem
        Caption = 'Rectang'
        OnClick = Rectang1Click
      end
    end
    object Color1: TMenuItem
      Caption = '&Color'
      object Red1: TMenuItem
        Caption = 'Red'
        OnClick = Red1Click
      end
      object Green1: TMenuItem
        Caption = 'Green'
        OnClick = Green1Click
      end
      object Blue1: TMenuItem
        Caption = 'Blue'
        OnClick = Blue1Click
      end
    end
    object Help1: TMenuItem...
    object About1: TMenuItem...
  end
end
```

I've listed the complete description of the menu items because their captions will play an important role in the code, as you'll see shortly. As a second step I've made the program work, by handling the various menu commands. Here are two examples from the two main pull-down menus:

```
procedure TForm1.Red1Click(Sender: TObject);
begin
  ShapeDemo.Brush.Color := clRed;
end;

procedure TForm1.Ellipse1Click(Sender: TObject);
```

```
begin
  ShapeDemo.Shape := stEllipse;
end;
```

I've also written a handler for the `OnResize` event of the form, to resize the shape depending on the actual size of the form. Instead of setting its four positional properties (`Left`, `Top`, `Width`, and `Height`) I've called the `SetBounds` method. This approach leads to faster code.

Now that we've written the code of the program, it is time to turn the menu items into bitmaps. To accomplish this I've prepared a bitmap file for each of the three shapes and one for each of the three base colors. Technically these bitmaps can have any size, but to make them look nice they should generally be wide and low (the typical size of a menu item).

Having prepared the six bitmap files, each named with the caption of its menu items (plus the `.BMP` extension), I've written a handler for the `OnCreate` event of the form, and replaced the text of the items of the fake standard menu with bitmaps. This can be accomplished by calling the `ModifyMenu` API function:

```
function ModifyMenu (hMnu: HMENU;
  uPosition, uFlags, uIDNewItem: UINT;
  lpNewItem: PChar): BOOL; stdcall;
```

The first parameter is the handle of the menu we are working on (typically a pull-down menu), the second is the position of the item we want to modify, the third is some menu flags indicating the effect of other parameters, the fourth is the identifier of the menu item (stored by Delphi in the `Command` property), and the last is the new string for the menu item. How can we use this to set a bitmap, instead? We simply pass the handle of a bitmap in the last parameter (instead of the string), and use the `mf_Bitmap` menu flag among the values of the `uFlags` parameter.

Now you are ready to look at the source code, which is based on two `for` loops. I wanted to make the code as generic as possible, so that adding a new kind of shape or color will be pretty straightforward:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  Bmp: TBitmap;
begin
  // load the bitmaps for the shapes
  for I := 0 to Shape1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Shape1.Items [I].Caption + '.bmp');
    ModifyMenu (Shape1.Handle,Shape1.Items [I].MenuIndex,
      mf_ByPosition or mf_Bitmap,
      Shape1.Items [I].Command, Pointer (Bmp.Handle));
    Shape1.Items [I].Tag := Integer (Bmp);
  end;

  // load the bitmaps for the colors
  for I := 0 to Color1.Count - 1 do
  begin
    Bmp := TBitmap.Create;
    Bmp.LoadFromFile (Color1.Items [I].Caption + '.bmp');
    ModifyMenu (Color1.Handle, Color1.Items [I].MenuIndex,
      mf_ByPosition or mf_Bitmap, Color1.Items [I].Command,
      Pointer (Bmp.Handle));
    Color1.Items [I].Tag := Integer (Bmp);
  end;
```

```
end;
```

> In the two calls to the `ModifyMenu` API function in the code above, we've used the `mf_ByPosition` flag in the third parameter, and passed the position of the item in the second parameter. However, we still need to pass the menu command as the fourth parameter, because this will be the command of the new menu item. If we don't, we'll lose the connection between the menu item and its Delphi event handler.

As you can see I save the reference to each bitmap object I create in the `Tag` property of the corresponding menu item. The only reason I have to save the bitmap objects is to be able to destroy them when the application terminates:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Shape1.Count - 1 do
    TBitmap (Shape1.Items [I].Tag).Free;
  for I := 0 to Color1.Count - 1 do
    TBitmap (Color1.Items [I].Tag).Free;
end;
```

As an alternative I could have saved the bitmap objects in an array (declared as a field of the form), and then destroyed each of the objects of the array at the end.

When you run this program, the Shape pull-down menu looks nice. The Color pull-down menu, however, is not working properly. As soon as you select it, in fact, the colors are displayed properly. But when you move over a menu item (as you can see by running the program) its colors are reversed by the menu item selection. Reversing the color you are asking for results in a very odd user interface: to select the color blue you have to click on an item that has temporarily turned to yellow!

Basically, you can use only black-and-white or gray-scaled bitmaps in a menu item. Color bitmaps create a lot of problems. If you want to obtain this effect, you should use an owner-draw menu item, as described in the next section.

Owner-Draw Menu Items

In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

Actually in Delphi the situation is slightly more complex. The components can take care of painting themselves also in this case (as is the case for the `TBitBtn` class for bitmap buttons), and eventually activate corresponding events. If you don't think about the internal details, owner-draw techniques for list boxes and combo boxes require you simply to write the handler for a couple of events.

Delphi provides no support for owner-draw menu items, though. So in this case we have to use the standard Windows approach, and handle a couple of system messages in our form. Here is the definition of these methods in the source code of the ODMenu example, an extension of the BitMenu example discussed in the last section:

```
type
```

```
  TForm1 = class(TForm)
    ...
  public
    procedure WmMeasureItem (var Msg: TWmMeasureItem);
      message wm_MeasureItem;
    procedure WmDrawItem (var Msg: TWmDrawItem);
      message wm_DrawItem;
  end;
```

The `wm_MeasureItem` message is sent by Windows once for each menu item when the pull-down menu is displayed to determine the size of each item. The `wm_DrawItem` message is sent when an item has to be repainted. This happens when Windows first displays the items, and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on.

In the ODMenu example I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've modified the `CreateForm` method to call the `ModifyMenu` API function passing the `mfOwnerDraw` menu flag. I also pass a code as the last parameter, to distinguish the various items. The other parameters are the same as in the previous version of the example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ...
  // turn the menu items into owner-draw items
  for I := 0 to Color1.Count - 1 do
  begin
    ModifyMenu (Color1.Handle, Color1.Items [I].MenuIndex,
      mf_ByPosition or mf_OwnerDraw,
      Color1.Items [I].Command, Pointer (I));
  end;
end;
```

Now we have to write the code of the two owner-draw message handlers. The `WmMeasureItem` method receives as a parameter (in the corresponding `TWmMeasureItem` structure) a pointer to the Windows `MeasureItemStruct` structure (you can see the details of this structure in the Windows API help file). From this last structure we use the `CtlType` field, which stores the type of element we are measuring, to check whether this operation pertains to a menu. If it does, we use the `ItemWidth` and `ItemHeight` fields to set the width and height of the item. Since all items have the same size, the code is quite simple. The last thing we have to do is to provide a return value for the message, indicating we have handled it, in the `Result` field of the message structure. Here is the complete code:

```
procedure TForm1.WmMeasureItem (var Msg: TWmMeasureItem);
begin
  inherited;
  with Msg.MeasureItemStruct^ do
    if CtlType = odt_Menu then
    begin
      ItemWidth := 80;
      ItemHeight := 30;
      Msg.Result := 1; // we've handled it
    end;
end;
```

Drawing a menu item is slightly more complex, since to write Delphi code we have to create a TCanvas object, which encapsulates a Windows *device context* handle. Since we have to free this object to avoid memory leaks, I've used a try-finally statement to destroy it even if an error occurs:

```
Canvas1 := TCanvas.Create;
Canvas1.Handle := hDC;
try
  ...
finally
  Canvas1.Free;
end;
```

The code assigns to the Handle property of this TCanvas object the hDC field passed by the message in the DrawItemStruct structure (there is, again, a pointer to this structure passed in one of the fields of the TWmDrawItem parameter of the message-handler method). This hDC field passes to our code the handle to the device context (the painting area, the Canvas) of the pull-down menu we are going to paint.

Once we have set up the drawing mechanism, we can start with the actual code. First we have to check the state of the menu item, stored in the ItemState field. If this includes the ods_Selected flag (a condition we can test by checking the result of a bitwise and expression); then we have to paint the background of the menu item using the Windows system color for the highlighted items, clHighlight. Otherwise, we use the standard color for menus, clMenu.

> Some of the Delphi constants for colors correspond to the Windows system color. These colors are not fixed, but reflect the current color setting made by the user.

Once we've assigned to the Brush of the canvas the background color, we can easily paint it by calling the FillRect method. This method has one single parameter, corresponding to the rectangle we have to erase. This information is available in the rcItem field of the DrawItemStruct structure. Notice, by the way, that if you don't limit your drawing area to this rectangle, you can paint over other menu items as well!

The second step is drawing the actual colored areas, which don't depend on the status of the menu item, since we want each color to show up properly even if the item is selected. By looking to the ItemData field, the code retrieves the code passed as the last parameter of the ModifyMenu function call, used to make the menu item owner-draw. Depending on the value of this field, the program sets a proper value for the Color of the Brush using a simple case statement. At this point we only have to paint the area, by calling the Rectangle function and passing, in the four parameters, a smaller area than the full surface of the menu item. In fact we need to reserve a border with the background color. In the example the border is 5 pixels wide.

Here, finally, is the complete source code of the WmDrawItem method:

```
procedure TForm1.WmDrawItem (var Msg: TWmDrawItem);
var
  Canvas1: TCanvas;
begin
  inherited;
  with Msg.DrawItemStruct^ do
    if CtlType = odt_Menu then
    begin
      // create a canvas for painting
      Canvas1 := TCanvas.Create;
      Canvas1.Handle := hDC;
      try
```

```
          // set the background color and draw it
          if (ods_Selected and ItemState <> 0) then
            Canvas1.Brush.Color := clHighlight
          else
            Canvas1.Brush.Color := clMenu;
          Canvas1.FillRect (rcItem);
          case ItemData of
            0: Canvas1.Brush.Color := clRed;
            1: Canvas1.Brush.Color := clLime;
            2: Canvas1.Brush.Color := clBlue;
          end;
          Canvas1.Rectangle (rcItem.Left + 5, rcItem.Top + 5,
            rcItem.Right - 10, rcItem.Bottom - 10);
        finally
          Canvas1.Free;
        end;
      end;
end;
```

I suggest you run this program along with the BitMenu program, to see the differences in the way the colored menu items are painted. Actually, looking at these two examples one might think of using the owner-draw technique also for menu items with black-and-white bitmaps.

Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and for "quick-and-dirty" applications. Adding a single menu item to the system menu is straightforward:

```
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_STRING, idSysAbout, '&About...');
```

The code fragment above (extracted from the SysMenu example) adds a separator and a new item to the system menu item. The GetSystemMenu API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The AppendMenu API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I've defined this identifier as:

```
const
  idSysAbout = 100;
```

In the SysMenu example, this code is executed in the OnCreate event handler, and it produces the new system menu. Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the wm_Command Windows message. This is handled internally by Delphi, which activates the OnClick event of the corresponding menu item component. The selection of system menu commands, instead, generates a wm_SysCommand message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.

We can intercept this command and check to see whether the command identifier (passed in the CmdType field of the TWmSysCommand parameter) of the menu item is our idSysAbout. Since there isn't a corresponding event in Delphi, we have to define a new message response method to the form class:

```
public
  procedure WMSysCommand (var Msg: TMessage);
    message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a MainMenu component to the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the Menu property of the form, removing the menu bar. This way we have a MainMenu component but nothing on the screen.

Now we can add some code to the SysMenu example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is pressed. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (self.Handle, FALSE),
        mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

This code uses the expression self.Handle to access the handle of the form. This is required because we are currently working on the MainMenu1 component, as specified by the with statement.

The menu flag used in this case, mf_Popup, indicates that we are adding a pull-down menu. In this function call the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example we passed the identifier of the menu, instead). Since we are adding to the system menu items with sub-menus, the final structure of the system menu will have two levels.

> The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms for two different things, the local menus and the secondary menus of the menu bar. Apparently, they've done this because these two elements are implemented with the same kind of internal windows; and the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course you can check for each menu item in the `WMSysCommand` method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the `OnClick` event of each item, as usual, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a `FindItem` method.

When we have found the menu item (and if we have found something), we can call its `Click` method (which invokes the `OnClick` handler). Here is the code I've added to the `WMSysCommand` method:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
  if Item <> nil then
    Item.Click;
```

In this code, The `CmdType` field of the message structure that is passed to the `WMSysCommand` procedure holds the command of the menu item being called.

You can also use a simple `if` or `case` statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as `sc_Close`, `sc_Minimize`, `sc_Maximize`, and so on. For more information, you can see the description of the `wm_SysCommand` message in the Windows API Help file, available in Delphi.

> This application works but has one glitch. If you click the right mouse button over the TaskBar icon representing the application, you get a plain system menu (actually even different than the default one). The reason is that this system menu belongs to a different window, the window of the `Application` global object.

Building a Complete Menu

Now that we know how to write a menu, disable and check menu items, and so on, we are ready to build the menu for a full-fledged application. Do you remember the RichNote example of the last chapter? It was a simple editor based on a RichEdit component. You could use it to write and change the font of the selected text, but that was all. Now we want to add a menu and implement a number of features, including a complete scheme for opening and saving the text files. In fact, we want to be able to ask the user to save any modified file before opening a new one, to avoid losing any changes. Sounds like a professional application, doesn't it?

First of all, we need to build the menu, following the standard. The main menu starts with two standard pull-down menus, File and Edit, with the typical menu items. Then there are two specific pull-down menus, Font and Paragraph, with menu items to set the text font and alignment. The last two pull-down menus, Options and Help, are *almost* standard: Their names are standard, but their menu items are not. The Options menu has commands to change the background color and to count the characters, and the Help menu has only the About menu item.

In this example, we want to implement most but not all of the commands of the menu, for example skipping the Clipboard commands. The following table shows the complete structure of the menu:

&File	&Edit	F&ont	&Paragraph	&Options	&Help
&New	Cu&t	&Times New Roman	&Left Aligned	&Backgroud Color	&About RichNote...
&Open...	&Copy	&Courier New	&Right Aligned	&Read Only	
&Save	&Paste	&Arial	&Centered	&Count chars...	
Save&As...		&Bold			
&Print...		&Italic			
E&xit		&Small			
		&Medium			
		&Large			
		More &Fonts...			

Having added a complete menu, we can now get rid of the simple panel and the font button of the RichNote example. The only visual component left in the form of the RichNot2 version will be the RichEdit component, which is aligned with the client area. Then there is the MainMenu component, and four components for standard dialog boxes (OpenDialog, SaveDialog, FontDialog, and ColorDialog).

The File Menu

As I mentioned when we began working through the RichNot2 example, the most complex part of this program is implementing the commands of the File pull-down menu—New, Open, Save, and Save As. In each case, we need to track whether the current file has changed, saving the file only if it has. We should prompt the user to save the file each time the program creates a new file, loads an existing one, or terminates.

To accomplish this, I've added two fields and three methods to the class describing the form of the application:

```
  private
    FileName: string;
    Modified: Boolean;
  public
    function SaveChanges: Boolean;
    function Save: Boolean;
    function SaveAs: Boolean;
```

The `FileName` string and the `Modified` flag are set when the form is created and changed when a new file is loaded or the user renames a file with the Save As command. These two flags are initialized when the form is first created:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
begin
  FileName := '';
  Modified := False;
end;
```

The value of the flag changes as soon as you type new characters in the RichEdit control (in its `OnChange` event handler):

```
procedure TFormRichNote.RichEdit1Change(Sender: TObject);
begin
  Modified := True;
end;
```

When a new file is created, the program checks whether the text has been modified. If so, it calls the SaveChanges function, which asks the user whether to save the changes, discard them, or skip the current operation:

```
procedure TFormRichNote.New1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
  begin
    RichEdit1.Text := '';
    Modified := False;
    FileName := '';
    Caption := 'RichNote - [Untitled]';
  end;
end;
```

If the creation of a new file is confirmed, some simple operations take place, including using *'Untitled'* instead of the file name in the form's caption.

Short-Circuit Evaluation

The expression **if not** Modified **or** SaveChanges **then** requires some explanation. By default, Pascal performs what is called "short-circuit evaluation" of complex conditional expressions. The idea is simple: if the expression not Modified is true, we are sure that the whole expression is going to be true, and we don't need to evaluate the second expression. In this particular case, the second expression is a function call, and the function is called only if Modified is True. This behavior of or and and expressions can be changed by setting a Delphi compiler option called Complete Boolean Eval. You can find it on the Compiler page of the Project Options dialog box.

The message box displayed by the SaveChanges function has three options. If the user selects the Cancel button, the function returns False. If the user selects No, nothing happens (the file is not saved) and the function returns True, to indicate that although we haven't actually saved the file, the requested operation (such as creating a new file) can be accomplished. If the user selects Yes, the file is saved and the function returns True.

In the code of this function, notice in particular the call to the MessageDlg function used as the value of a case statement:

```
function TFormRichNote.SaveChanges: Boolean;
begin
  case MessageDlg (
    'The document ' + filename + ' has changed.' +
    #13#13 + 'Do you want to save the changes?',
    mtConfirmation, mbYesNoCancel, 0) of
  idYes:
    // call Save and return its result
    Result := Save;
  idNo:
    // don't save and continue
    Result := True;
  else // idCancel:
    // don't save and abort operation
    Resulht := False;
  end;
```

```
end;
```

> In the MessageDlg call above, I've added explicit newline characters (`#13`) to improve the readability of the output. As an alternative to using a numeric character constant, you can call `Chr(13)`.

To actually save the file, another function is invoked: `Save`. This method saves the file if it already has a proper file name or asks the user to enter a name, calling the `SaveAs` functions. These are two more internal functions, not directly connected with menu items:

```
function TFormRichNote.Save: Boolean;
begin
  if Filename = '' then
    Result := SaveAs // ask for a file name
  else
  begin
    RichEdit1.Lines.SaveToFile (FileName);
    Modified := False;
    Result := True;
  end;
end;

function TFormRichNote.SaveAs: Boolean;
begin
  SaveDialog1.FileName := Filename;
  if SaveDialog1.Execute then
  begin
    Filename := SaveDialog1.FileName;
    Save;
    Caption := 'RichNote - ' + Filename;
    Result := True;
  end
  else
    Result := False;
end;
```

I use two functions to perform the Save and SaveAs operations (and do not call the corresponding menu handler directly) because I need a way to report a request to cancel the operation from the user. To avoid code duplication, the handlers of the Save and SaveAs menu items call the two functions too, although they ignore the return value:

```
procedure TFormRichNote.Save1Click(Sender: TObject);
begin
  if Modified then
    Save;
end;

procedure TFormRichNote.Saveas1Click(Sender: TObject);
begin
  SaveAs;
end;
```

Opening a file is much simpler. Before loading a new file, the program checks whether the current file has changed, asking the user to save it with the `SaveChanges` function, as before. The `Open1Click` method is based on the OpenDialog component, another default dialog box provided by Windows and supported by Delphi:

```
procedure TFormRichNote.Open1Click(Sender: TObject);
begin
  if not Modified or SaveChanges then
    if OpenDialog1.Execute then
    begin
      Filename := OpenDialog1.FileName;
      RichEdit1.Lines.LoadFromFile (FileName);
      Modified := False;
      Caption := 'RichNote - ' + FileName;
    end;
end;
```

The only other detail related to file operations is that both the OpenDialog and SaveDialog components of the NotesForm have a particular value for their `Filter` and `DefaultExt` properties, as you can see in the following fragment from the textual description of the form:

```
object OpenDialog1: TOpenDialog
  DefaultExt = 'rtf'
  FileEditStyle = fsEdit
  Filter = 'Rich Text File (*.rtf)|*.rtf|Any file (*.*)|*.*'
  Options = [ofHideReadOnly, ofPathMustExist,ofFileMustExist]
end
```

The string used for the `Filter` property (which should be written on a single line) contains four pairs of substrings, separated by the | symbol. Each pair has a description of the type of file that will appear in the File Open or File Save dialog box, and the filter to be applied to the files in the directory, such as `*.RTF`. To set the filters in Delphi, you can simply invoke the editor of this property, which displays a list with two columns.

The file-related methods above are also called from the `FormCloseQuery` method (the handler of the `OnCloseQuery` event), which is called each time the user tries to close the form, terminating the program. We can make this happen in various ways — by double-clicking on the system menu icon, selecting the system menu's Close command, pressing the Alt+F4 keys, or calling the `Close` method in the code, as in the File | Exit menu command.

In `FormCloseQuery`, you can decide whether or not to actually close the application by setting the `CanClose` parameter, which is passed by reference. Again, if the current file has been modified, we call the `SaveChanges` function and use its return value. Again we can use the short-circuit evaluation technique:

```
procedure TFormRichNote.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not Modified or SaveChanges;
end;
```

The last menu item of the File menu is the Print command. Since the RichEdit component includes print capabilities and they are very simple to use, I've decided to implement it anyway. Here is the code, which actually produces a very nice printout:

```
procedure TFormRichNote.Print1Click(Sender: TObject);
begin
  RichEdit1.Print (FileName);
end;
```

The Paragraph Menu

Compared to the File menu, the other pull-down menus of this example are simpler. The code of the Paragraph menu is based on some properties of its items. Here is their textual description (from the DFM file):

```
object Paragraph1: TMenuItem
  Caption = '&Paragraph'
  object LeftAligned1: TMenuItem
    Caption = '&Left Aligned'
    Checked = True
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
  object RightAligned1: TMenuItem
    Caption = '&Right Aligned'
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
  object Centered1: TMenuItem
    Caption = '&Centered'
    GroupIndex = 1
    RadioItem = True
    OnClick = RightAligned1Click
  end
end
```

As you can see, the program uses radio menu items, by giving to the three items the same value for the `GroupIndex` property and setting the `RadioItem` property to `True`. The menu items also share the same `RightAligned1Click` method for their `OnClick` event. Here is the code of the method, which is based on the correspondence between the position of the menu items in the pull-down (indicated by their `MenuIndex` property) and the order of the values of the `TAlignment` enumeration. It is a trick, but it works. Here is the code:

```
procedure TFormRichNote.RightAligned1Click(Sender: TObject);
begin
  RichEdit1.Paragraph.Alignment :=
    TAlignment ((Sender as TMenuItem).MenuIndex);
  (Sender as TMenuItem).Checked := True;
end;
```

First, this procedure sets the alignment of the current paragraph (the paragraph including the selected text or the editor cursor), then it checks the current menu item — the menu item that has activated the method (the `Sender` object). As you can see, this code relies on some controlled typecasts, based on the `as` keyword: this is what you have to do any time you want to write generic code (that is, to attach the same methods to events of different components).

Notice that setting the check mark for the current menu item is correct only until you change the selection or the current line in the text. For this reason, we can handle the `OnSelectionChange` event of the RichEdit component, and update the check mark of the Paragraph menu each time:

```
procedure TFormRichNote.RichEdit1SelectionChange(Sender: TObject);
begin
  Paragraph1.Items [Integer (RichEdit1.Paragraph.Alignment)].
    Checked := True;
```

```
end;
```

The Font Menu

The Font pull-down menu is built on the same concept, but it has two groups of radio items, plus items with a standard check mark. Each group, then, uses a different approach to handle the menu item selection with a single event response method. You can see the details of the properties of the menu items directly in the `RichForm.DFM` source code file (this listing was too long to reproduce here).

The code of the first group of menu items (used to select the font) is based on a simple trick: the name of the font to select corresponds to the `Caption` of the menu item, without the initial `&` character:

```
procedure TFormRichNote.TimesRoman1Click(Sender: TObject);
var
  FontName: string;
begin
  // get the font name and remove the &
  FontName := (Sender as TMenuItem).Caption;
  Delete (FontName, 1, 1);
  // change selected text font
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Name := FontName;
  (Sender as TMenuItem).Checked := True;
end;
```

This code acts on the current selection (using the `SelAttributes` property of the RichEdit1 component), as the RichNote example in Chapter 8 did. Notice that you can easily extend this code by adding new menu items consisting of the name of the font preceded by the `&` character. Then you'll necessarily have to set the same value for the `GroupIndex` property of the previous items (in this case `1`). So if you simply set the `RadioItem` property to `True` and connect the `OnClick` event to the `TimesRoman1Click` method, they'll behave just like the existing font selection menu items.

The second group of items controls the selection of the bold and italic styles. This is accomplished by two similar but separate methods. Here is one of them:

```
procedure TFormRichNote.Bold1Click(Sender: TObject);
begin
  Bold1.Checked := not Bold1.Checked;
  if RichEdit1.SelLength > 0 then
    with RichEdit1.SelAttributes do
      if Bold1.Checked then
        Style := Style + [fsBold]
      else
        Style := Style - [fsBold];
end;
```

The last part of the menu has another group of radio menu items, used to set the size of the font. These menu items refer to a font size in the caption and have the same value stored in their `Tag` property. This makes the code very easy to write. Again, there is a single method for the three menu items, but you can add new items to this group with very little effort:

```
procedure TFormRichNote.Large1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
    RichEdit1.SelAttributes.Size :=(Sender as TMenuItem).Tag;
  (Sender as TMenuItem).Checked := True;
end;
```

The last item of this menu simply activates the Font dialog box. Notice that the font returned by this dialog box cannot be assigned directly to the `SelAttributes` property; we need to call the `Assign` method, instead:

```
procedure TFormRichNote.More1Click(Sender: TObject);
begin
  FontDialog1.Font := RichEdit1.Font;
  if FontDialog1.Execute then
    RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  // update the check marks
  RichEdit1SelectionChange (self);
end;
```

All the commands of this menu affect the status of the current selection, setting the check boxes and radio items properly. The method above, instead, calls the `RichEdit1SelectionChange` method. This is the handler of the `OnSelectionChange` event of the RichEdit component.

This method scans some of the groups to determine which element has to be checked. As in other examples before, this code has been written so that you can easily extend it for new menu items (the listing also includes the code used to reset the paragraph alignment, discussed earlier):

```
procedure TFormRichNote.RichEdit1SelectionChange(Sender: TObject);
var
  FontName: string;
  I: Integer;
begin
  // check the font name radio menu item
  FontName := '&' + RichEdit1.SelAttributes.Name;
  for I := 0 to 2 do
    with Font1.Items [I] do
      if FontName = Caption then
        Checked := True;

  // check the bold and italic items
  Italic1.Checked :=
    fsItalic in RichEdit1.SelAttributes.Style;
  Bold1.Checked :=
    fsBold in RichEdit1.SelAttributes.Style;

  // check the font size
  for I := Small1.MenuIndex to Large1.MenuIndex do
    with Font1.Items [I] do
      if Tag = RichEdit1.SelAttributes.Size then
        Checked := True;

  // check the paragraph style
  Paragraph1.Items [Integer (RichEdit1.Paragraph.Alignment)].
    Checked := True;
end;
```

This method doesn't work perfectly. When you set a custom font, the selection doesn't change properly (because the current item remains selected). To fix it, we should remove any radio check mark when the value is not one of the possible selections, or add new menu items for this special case. The example is complex enough, so I think we can live with this minor inconvenience.

The Options Menu

The last pull-down menu of the RichNot2 example is the Options menu. This menu has three unrelated commands used to customize the user interface, and to determine and display the length of the text. The first command displays a color selection dialog box, used to change the color of the background of the RichEdit component:

```
procedure TFormRichNote.BackColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := RichEdit1.Color;
  if ColorDialog1.Execute then
    RichEdit1.Color := ColorDialog1.Color;
end;
```

The second command can be used to mark the text as read-only. In theory this property should be set depending on the status of the file on the disk. In the example, instead, the user can toggle the read-only attribute manually:

```
procedure TFormRichNote.ReadOnly1Click(Sender: TObject);
begin
  RichEdit1.ReadOnly := not RichEdit1.ReadOnly;
  ReadOnly1.Checked := not ReadOnly1.Checked;
end;
```

The last menu item activates a method to count the number of characters in the text and display the total in a message box. The core of the method is the call to the `GetTextLen` function of the RichEdit control. The number is extracted and formatted into an output string:

```
procedure TFormRichNote.Countchars1Click(Sender: TObject);
begin
  MessageDlg (Format (
    'The text has %d characters', [RichEdit1.GetTextLen]),
    mtInformation, [mbOK], 0);
end;
```

The handler of the `OnClick` event of this last item of the Options menu terminates the example. As mentioned at the beginning, this was a rather long and complex example, but its purpose was to show you the implementation of the menu commands of a real-world application. In particular, I explained in detail the File pull-down menu because this is something you'll probably need to handle in any file-related application.

Now we are ready to delve into another topic involving menus, the use of local menus activated by the right mouse button click.

Pop-Up Menus

In Windows, it is common to see applications that have special local menus you activate by clicking the right mouse button. The menu that is displayed — a pop-up menu, in common Windows terminology — usually depends on the position of the mouse click. These menus tend to be easy to use since they group only the few commands related to the element that is currently selected. They are also usually faster to use than full-blown menus because you don't need to move the mouse up to the menu bar and then down again to go on working.

In Delphi, there are basically two ways to display pop-up menus, using the corresponding component. You can let Delphi handle them automatically or you can choose a manual technique. I'll explore both approaches, starting with the first, which is the simplest one.

To add a pop-up menu to a form, you need to perform a few simple operations. Create a PopupMenu component, add some menu items to it, and select the component as the value of the form's `PopupMenu` property. That's all. Of course, you should also add some handlers for the `OnClick` events of the local menu's various menu items, as you do with an ordinary menu.

An Automatic Local Menu

To show you how to create a local menu, I've built an example that is an extension of the Dragging example. The new example is named Local1. I've added a first PopupMenu component to its form and connected it using the `PopupMenu` property of the form itself. Once this is done, running the program and clicking the right mouse button on the form displays the local menu. Then, I've added a second pop-up menu component, with two levels, to the form, and I've attached it to the StaticText component on the right, `LabelTarget`. To connect a local menu to a specific component, you simply need to set its `PopupMenu` property. The four methods related to the first group of commands of the Colors pull-down menu just select a color:

```
procedure TDraggingForm.Aqua1Click(Sender: TObject);
begin
  LabelTarget.Color := clAqua;
end;
```

The Transparent command selects the color of the parent form as the current color, setting the value of the `ParentColor` property to `True`.

> The components of a form usually borrow some properties from the form. This is indicated by specific properties, such as `ParentColor` or `ParentFont`. When these properties are set to `True`, the current value of the component's property is ignored, and the value of the form is used instead. Usually, this is not a problem, because as soon as you set a property of the component (for example, the font), the corresponding property indicating the use of the parent attribute (`ParentFont`) is automatically set to `False`.

The last command of the pull-down menu, User Defined, presents the standard Color Selection dialog box to the user. The three commands of the pop-up menu's second pull-down change the alignment of the text of the big label and add a check mark near the current selection, deselecting the other two menu items. Here is one of the three methods:

```
procedure TDraggingForm.Center1Click(Sender: TObject);
begin
  LabelTarget.Alignment := taCenter;
  Left1.Checked := False;
  Center1.Checked := True;
  Right1.Checked := False;
end;
```

A pop-up menu, in fact, can use all the features of a main menu and can have checked, disabled, or hidden items, and more. I could have also used radio menu items for this pop-up menu, but this doesn't seem to be a very common approach.

Modifying a Pop-Up Menu When It Is Activated

Why not use the same technique to display a check mark near the selected color? It is possible, but it's not a very good solution. In fact, there are six menu items to consider, and the color can also change when a user drags it from one of the labels on the left of the form. For this reason, and to show you another technique, I've followed a different approach.

Each time a pop-up menu is displayed, the `OnPopup` event is sent to your application. In the code of the corresponding method, you can place the check mark on the current selection of the color, independently from the action used to set it:

```
procedure TDraggingForm.PopupMenu2Popup(Sender: TObject);
var
  I: Integer;
begin
  // unchecks all menu items (not required for radio menu items)
  with Colors1 do
    for I := 0 to Count - 1 do
      Items[I].Checked := False;

  // checks the proper item
  case LabelTarget.Color of
    clRed: Red1.Checked := True;
    clAqua: Aqua1.Checked := True;
    clGreen: Green1.Checked := True;
    clYellow: Yellow1.Checked := True;
  else
    if LabelTarget.ParentColor then
      Transparent1.Checked := True
    else
      UserDefined1.Checked := True;
  end;
end;
```

This method's code requires some explanation. At the beginning, the menu items are all unchecked by using a `for` loop on the `Items` array of the Colors1 menu. The advantage of this loop is that it operates on all the menu items, regardless of their number. Then the program uses a `case` statement to check the proper item.

Handling Pop-Up Menus Manually

In the Local1 example, we saw how to use automatic pop-up menus. As an alternative, you can set the `AutoPopup` property to `False` or not connect the pop-up menu to any component, and use the pop-up menu's `Popup` method to display it on the screen. This procedure requires two parameters: the *x* and *y* values of the position where the menu is going to be displayed. The problem is that you need to supply the *screen* coordinates of the point, not the client coordinates, which are the usual coordinates relative to the form's client area.

As an example, I've taken an existing application with a menu — the third version of the MenuOne example, described in this chapter — and added a peculiar pop-up menu. The idea is that there are two different pop-up menus, one to change the colors and the other to change the alignment of the text. Each time the user right-clicks on the caption, one of the two pop-up menus is displayed. In real applications, you'll probably have to decide which menu to display depending on the status of some variable. Here, I've followed a simple (and arbitrary) rule: Each time the right mouse button is clicked, the pop-up menu changes. My aim is to show you how to do this in the simplest possible way.

The two pop-up menus are very simple, and correspond to actions already available in the main menu (and connected with the same event handlers). Actually, I've built these pop-up menus by copying the main menu to a menu template and then pasting from it. The only change I've made is to remove the shortcut keys.

When the user clicks the right mouse button over the label, which takes up the whole surface of the form, a method displays one of the two menus. Instead of using the `OnClick` event, I've trapped the `OnMouseDown` event of the label. This second event passes as parameters the coordinates of the mouse click.

These coordinates are relative to the label, so you have to convert them to screen coordinates by calling the `ClientToScreen` method of the label:

```
procedure TFormColorText.Label1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ClientPoint, ScreenPoint: TPoint;
begin
  if Button = mbRight then
  begin
    ClientPoint.X := X;
    ClientPoint.Y := Y;
    ScreenPoint := Label1.ClientToScreen (ClientPoint);
    Inc (ClickCount);
    if Odd (ClickCount) then
      PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
    else
      PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  end;
end;
```

In this procedure, you first have to check whether the right mouse button was clicked. The second step is to translate the coordinate of the position of the mouse click from client coordinates to screen coordinates. Screen coordinates are required by the PopupMenu component's `Popup` method.

The last thing we have to do is provide the proper check marks for the menu items of the second pop-up menu. A solution is to copy the current check marks of the main menu to the pop-up before displaying it:

```
if Odd (ClickCount) then
  PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
else
begin
  {set the check marks as in the main menu}
  Left2.Checked := Left1.Checked;
  Center2.Checked := Center1.Checked;
  Right2.Checked := Right1.Checked;
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
end;
```

An alternative solution—the one I've actually implemented in the Local2 example — is to set this check mark every time you set the check marks of the main menu, as in the following code:

```
procedure TFormColorText.Left1Click(Sender: TObject);
begin
  Label1.Alignment := taLeftJustify;
  Left1.Checked := True;
  Left2.Checked := True;
end;
```

In a more general application you might want to write a generic routine to apply the check marks more consistently, but in this example one of these two simple techniques will do.

What’s Next

In this chapter, we have seen how to create main menus and pop-up menus in Delphi. We’ve discussed the standard guidelines for the names of the pull-down menus and of the menu items, shortcut keys, check marks, graphical menus, local menus, and many other topics. You can explore in other directions, as well. For example, you can create a menu dynamically (at run-time) or copy portions of a menu to another menu, as in the SysMenu example.

The next step, however, is to explore a less common but very nice feature of Delphi programming, multimedia.

CHAPTER 6: MULTIMEDIA FUN

- Windows default sounds
- From a beep to music
- The Media Player component
- Playing sounds and running videos
- Applications for audio CD drives

Among the many devices, the multimedia subsystem focuses sound cards or CD-ROM drives. Of course, besides being physically connected to your computer, these devices must be properly installed in Windows for your Delphi applications to access them.

Windows provides a specific API, known as the *Multimedia API*, to handle external devices such as video, MIDI, and CD drives. Delphi includes a corresponding Help file along with an easy-to-use component, the Media Player, to manipulate most multimedia devices. Before discussing this component, which is the main topic of the chapter, we'll look at some simpler ways to produce sound in Windows, beyond the simple beeps we have used previously.

Windows Default Sounds

In the book's earlier examples, every time we wanted to notify the user of an error or a specific event, we called a Delphi system procedure (`Beep`) or a Windows API function (`MessageBeep`). The `Beep` procedure is defined in the Delphi run-time library as follows:

```
procedure Beep;
begin
  MessageBeep(0);
end;
```

It simply passes the value 0 to the `MessageBeep` API function. Besides the values 0 and `-1`, both used to produce a beep with the internal speaker of the computer, the `MessageBeep` function can also accept other values, and play the corresponding sounds with your sound board. Here are the acceptable constants and the corresponding Windows sounds they produce (these are the sound names available in Control Panel):

`mb_IconAsterisk`	SystemAsterisk sound
`mb_IconExclamation`	SystemExclamation sound
`mb_IconHand`	SystemHand sound
`mb_IconQuestion`	SystemQuestion sound
`mb_Ok`	SystemDefault sound

You can change the association between system events and sound files using the Control Panel, which lists the sounds under the names shown in the right column above. These associations are stored in the Windows System Registry.

Notice that these constants are also the possible values of the MessageBox API function, encapsulated in the MessageBox method of the TApplication class. It is common to produce the corresponding sound when the message box is displayed. This feature is not directly available in the Delphi MessageDlg function (which displays a message box with a corresponding icon), but we can extend it easily by building a SoundMessageDlg function, as demonstrated by the following example.

Every Box Has a Beep

To show you the capabilities of the MessageBeep API function, I've prepared a simple example, Beeps. The form of this example has a RadioGroup with some radio buttons from which the user can choose one of the five valid constants of the MessageBeep function. Here is the definition of the RadioGroup, from the textual description of the form:

```
object RadioGroup1: TRadioGroup
  Caption = 'Parameters'
  ItemIndex = 0
  Items.Strings = (
    'mb_IconAsterisk'
    'mb_IconExclamation'
    'mb_IconHand'
    'mb_IconQuestion'
    'mb_Ok')
end
```

The program plays the sound corresponding to the current selection when the user clicks on the Beep Sound button (one of the push buttons of the form). This button's OnClick event-handler first determines which radio button was selected, using a case statement, and then plays the corresponding sound:

```
procedure TForm1.BeepButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
    4: BeepConstant := mb_Ok;
  else
    BeepConstant := 0;
  end;
  MessageBeep (BeepConstant);
end;
```

The else clause of the case statement is provided mainly to prevent an annoying (but not dangerous) compiler warning. To compare the selected sound with the default beep sound, click on the second button of the column (labeled *Beep –1*), which has the following code:

```
procedure TForm1.BeepOneButtonClick(Sender: TObject);
begin
  MessageBeep (Cardinal (-1));
end;
```

You can also pass the corresponding $FFFFFFFF hexadecimal value to the MessageBeep function. There is actually no difference between the two approaches. To test whether a sound driver is installed in your system (with or without a sound card, since it is possible to have a sound driver for the PC speaker), click on the first button (labeled *Test*), which uses a multimedia function, `WaveOutGetNumDevs`, to perform the test:

```
procedure TForm1.TestButtonClick(Sender: TObject);
begin
  if WaveOutGetNumDevs > 0 then
    SoundMessageDlg ('Sound is supported',
      mtInformation, [mbOk], 0)
  else
    SoundMessageDlg ('Sound is NOT supported',
      mtError, [mbOk], 0);
end;
```

To compile this function, you need to add the MmSystem unit to the `uses` clause. If your computer has no sound driver installed, you will hear only standard beeps, regardless of which sound is selected. The last two buttons have a similar aim: they both display a message box and play the corresponding sound.

The `OnClick` event handler of the Message Box button uses the traditional Windows approach. It calls the `MessageBeep` function and then the `MessageBox` method of the `Application` object soon afterward. The effect is that the sound is 1played when the message box is displayed. In fact (depending on the sound driver), playing a sound doesn't usually stop other Windows operations. Here is the code related to this fourth button:

```
procedure TForm1.BoxButtonClick(Sender: TObject);
var
  BeepConstant: Cardinal;
begin
  case RadioGroup1.ItemIndex of
    0: BeepConstant := mb_IconAsterisk;
    1: BeepConstant := mb_IconExclamation;
    2: BeepConstant := mb_IconHand;
    3: BeepConstant := mb_IconQuestion;
  else {including 4:}
    BeepConstant := mb_Ok;
  end;
  MessageBeep (BeepConstant);
  Application.MessageBox (
    PChar (RadioGroup1.Items [RadioGroup1.ItemIndex]),
    'Sound', BeepConstant);
end;
```

If you click on the last button, the program calls the `SoundMessageDlg` function, which is not an internal Delphi function. It's one I've added to the program, but you can use it in your applications. The only suggestion I have is to choose a shorter name if you want to use it frequently. `SoundMessageDlg` plays a sound, specified by its `AType` parameter, and then displays the Delphi standard message box:

```
function SoundMessageDlg (const Msg: string;
  AType: TMsgDlgType; AButtons: TMsgDlgButtons;
  HelpCtx: Longint): Integer;
var
  BeepConstant: Cardinal;
begin
  case AType of
    mtWarning: BeepConstant := mb_IconExclamation;
    mtError: BeepConstant := mb_IconHand;
    mtInformation: BeepConstant := mb_IconAsterisk;
    mtConfirmation: BeepConstant := mb_IconQuestion;
  else
    BeepConstant := mb_Ok;
  end;
  MessageBeep(BeepConstant);
  Result := MessageDlg (Msg, AType,
    AButtons, HelpCtx);
end;

procedure TForm1.MessDlgButtonClick(Sender: TObject);
var
  DlgType: TMsgDlgType;
begin
  case RadioGroup1.ItemIndex of
    0: DlgType := mtInformation;
    1: DlgType := mtWarning;
    2: DlgType := mtError;
    3: DlgType := mtConfirmation;
  else {including 4:}
    DlgType := mtCustom;
  end;
  SoundMessageDlg (
    RadioGroup1.Items [RadioGroup1.ItemIndex],
    DlgType, [mbOK], 0);
end;
```

SoundMessageDlg is a simple function, but your programs can really benefit from its use.

From Beeps to Music

When you use the MessageBeep function, your choice of sounds is limited to the default system sounds. Another Windows API function, PlaySound, can be used to play a system sound, as well as any other waveform file (WAV). Again, I've built a simple example to show you this approach. The example is named ExtBeep (for Extended Beep) and has the simple form.

The form's list box shows the names of some system sounds and some WAV files, available in the current directory (that is, the directory containing ExtBeep.exe itself). When the user clicks on the Play button, the PlaySound function (defined in the MmSystem unit) is called:

```
procedure TForm1.PlayButtonClick(Sender: TObject);
begin
  PlaySound (PChar (Listbox1.Items [ListBox1.ItemIndex]),
    0, snd_Async);
end;
```

The first parameter is the name of the sound—either a system sound, a WAV file, or a specific sound resource (see the Win32 API Help file for details). The second parameter specifies where to look for a resource sound, and the third contains a series of flags, in this case indicating that the function should return immediately and let the sound play asynchronously. (An alternative value for this parameter is snd_Sync. If you use this value, the function won't return until the sound has finished playing.) With asynchronous play, you can interrupt a long sound by calling the PlaySound function again, using nil for the first parameter:

```
procedure TForm1.StopButtonClick(Sender: TObject);
begin
  PlaySound (nil, 0, 0);
```

```
end;
```

This is the code executed by the ExtBeep example when the user clicks on the Stop button. This button is particularly useful for stopping the repeated execution of the sound started by the Loop button, which calls `PlaySound` passing as its last parameter `(snd_Async or snd_Loop)`.

The only other method of the example, `FormCreate`, selects the first item of the list box at startup, by setting its `ItemIndex` property to 0. This avoids run-time errors if the user clicks on the button before selecting an item from the list box. You can test this example by running it, and by adding the names of the other `WAV` files or system sounds (as listed in the registration database). I suggest you also test other values for the third parameter of the function (see the API help files for details).

The Media Player Component

Now let's move back to Delphi and use the Media Player component. The Delphi `TMediaPlayer` class encapsulates most of the capabilities of the Windows Media Control Interface (MCI), a high-level interface for controlling internal and external media devices.

Perhaps the most important property of the `TMediaPlayer` component is `DeviceType`. Its value can be `dtAutoSelect`, indicating that the type of the device depends on the file extension of the current file (the `FileName` property). As an alternative, you can select a specific device type, such as `dtAVIVideo`, `dtCDAudio`, `dtWaveAudio`, and many others.

Once the device type (and eventually the file) have been selected, you can open the corresponding device (or set `AutoOpen` to `True`), and the buttons of the Media Player component will be enabled. The component has a number of buttons, not all of which are appropriate for each media type. There are actually three properties referring to the buttons: `VisibleButtons`, `EnabledButtons`, and `ColoredButtons`. The first determines which of the buttons are present in the control, the second determines which buttons are enabled, and the third determines which buttons have colored marks. By using the first two of these properties, you can permanently or temporarily hide or disable some of the buttons.

The component has several events. The `OnClick` event is unusual because it contains one parameter indicating which button was pressed and a second parameter you can use to disable the button's default action. The `OnNotify` event later tells the component whether the action generated by the button was successful. Another event, `OnPostClick`, is sent either when the action starts or when it ends, depending on the value of the `Wait` property. This property determines whether the operation on the device should be synchronous.

Playing Sound Files

Our first example using the Media Player is very simple. The form of the MmSound example has some labels describing the current status, a button to select a new file, an OpenDialog component, and a Media Player component with the following settings:

```
object MediaPlayer1: TMediaPlayer
  VisibleButtons = [btPlay, btPause,
    btStop, btNext, btPrev]
  OnClick = MediaPlayer1Click
  OnNotify = MediaPlayer1Notify
```

```
end
```

When a user opens a new file, a wave table, or a MIDI file, the program enables the Media Player, and you can play the sound and use the other buttons, too:

```
procedure TForm1.NewButtonClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    FileLabel.Caption := OpenDialog1.Filename;
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.Notify := True;
  end;
end;
```

Since I set the `Notify` property to `True`, the Media Player invokes the corresponding event handler, which outputs the information to a label:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  case MediaPlayer1.NotifyValue of
    nvSuccessful : NotifLabel.Caption := 'Success';
    nvSuperseded : NotifLabel.Caption := 'Superseded';
    nvAborted    : NotifLabel.Caption := 'Aborted';
    nvFailure    : NotifLabel.Caption := 'Failure';
  end;
  MediaPlayer1.Notify := True;
end;
```

Notice that you need to set the `Notify` property to `True` every time the `OnNotify` event handler is called in order to receive further notifications. Another label is updated to display the requested command.

```
procedure TForm1.MediaPlayer1Click(Sender: TObject;
  Button: TMPBtnType; var DoDefault: Boolean);
begin
  case Button of
    btPlay: ActionLabel.Caption := 'Playing';
    btPause: ActionLabel.Caption := 'Paused';
    btStop: ActionLabel.Caption := 'Stopped';
    btNext: ActionLabel.Caption := 'Next';
    btPrev: ActionLabel.Caption := 'Previous';
  end;
end;
```

Running Videos

So far, we have worked with sound only. Now it is time to move to another kind of media device: video. You indeed have a video device on your system, but to play video files (such as `AVI` files), you need a specific driver (directly available in Windows). If your computer can display videos, writing a Delphi application to do so is almost trivial: place a Media Player component in a form, select an `AVI` file in the `FileName` property, set the `AutoOpen` property to `True`, and run the program. As soon as you click on the Play button, the system opens a second window and shows the video in it.

Instead of playing the file in its own window, we can add a panel (or any other windowed component) to the form and use the name of this panel as the value of the Media Player's `Display` property. As an alternative, we can set the `Display` and the `DisplayRect` properties to indicate which portions of the output window the video should cover.

Although it is possible to create a similar program writing no code at all, to do so I would have to know which `AVI` files reside on your computer, and specify the full path of one of them in the `FileName` property of the Media Player component. As an alternative, I've written a simple routine to open and start playing a file automatically. You only have to click on the panel (as the caption suggests).

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.FileName := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.Perform (wm_LButtonDown, 0, $00090009);
    MediaPlayer1.Perform (wm_LButtonUp, 0, $00090009);
  end;
end;
```

After opening the Media Player, I could have called its `Play` method immediately to start it. But that would not have enabled and disabled the buttons properly. So I decided to simulate a click in position 9 on the *x*-axis and 9 on the *y*-axis of the Media Player window (instead of building the 32-bit value including both coordinates with a function, you can use the hexadecimal value directly, as in the code above). To avoid errors, I disabled all the buttons at design-time, until the simulated click takes place. I also automatically close the player when the application is closed (in the `OnClose` event handler).

A Video in a Form

The Media Player component has some limits regarding the window it can use to produce the output. You can use many components, but not all of them. A strange thing you can try is to use the Media Player component itself as the video's output window. This works, but there are two problems. First, the Media Player component cannot be aligned, and it cannot be sized at will. If you try to use big buttons, their size will be reduced automatically at run-time. The second problem is that if you click on the Pause button, you'll see the button in front of the video, while the other buttons are still covered. (I suggest you try this approach, anyway, just for fun.)

One thing you cannot do easily is display the video in a form. In fact, although you cannot set the form as the value of the Media Player's `Display` property at design-time, you can set it at run-time. To try this, simply place a hidden Media Player component (set the `Visible` property to `False`) and an `OpenDialog` component in a form. Set a proper title and hint for the form itself, and enable the `ShowHints` property. Then write the following code to load, start, and stop the video when the user clicks on the form:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  if MediaPlayer1.FileName = '' then
    if OpenDialog1.Execute then
    begin
      MediaPlayer1.FileName := OpenDialog1.FileName;
      MediaPlayer1.Open;
      Playing := False;
    end
    else
      exit; // stop if no file is selected
  if Playing then
  begin
    MediaPlayer1.Stop;
    Playing := False;
    Caption := 'MM Video (Stopped)';
    Hint := 'Click to play video';
  end
  else
  begin
    MediaPlayer1.Display := self;
    MediaPlayer1.DisplayRect := ClientRect;
    MediaPlayer1.Play;
    Playing := True;
    Caption := 'MMV (Playing)';
    Hint := 'Click to stop video';
  end;
end;
```

In this code, `Playing` is a private `Boolean` field of the form. Notice that the program shows the video using the full client area of the form. If the form is resized, you can simply enlarge the output rectangle accordingly:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  MediaPlayer1.DisplayRect := ClientRect;
end;
```

The best way to view a video is to use its original size, but with this program you can actually stretch it, and even change its proportions. Of course, the Media Player can also stop when it reaches the end of a file or when an error occurs. In both cases, we receive a notification event:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  Playing := False;
  Caption := 'MMV (Stopped)';
  Hint := 'Click to play video';
end;
```

Working with a CD Drive

In addition to audio and video files, the MCI interface is generally used to operate external devices. There are many examples, but the most common MCI device connected to a PC is probably a CD-ROM drive. Most CD-ROM drives can also read audio CDs, sending the output to an external speaker or a sound card. You can use the MCI interface and the Media Player component to write applications that handle such a device. Basically, you need to set the `DeviceType` property to `dtCDAudio`, making sure no file is selected in the `FileName` property, and be ready with a CD player.

In fact, just by placing a Media Player component in a form, setting the above properties, and compiling and running the program, you end up with a fully functional audio CD player. When you start customizing the player, though, not everything is as simple as it seems at first glance. I've built an example using some more capabilities of this component and of Windows multimedia support related to audio CDs. The form of this program has a couple of buttons, some labels to show the current status, a timer, and a SpinEdit component you can use to choose a track from the disk.

The idea is to use the labels to inform the user of the number of tracks on a disk, the current track, the current position within a track, and the length of the track, monitoring the current situation using the timer.

In general, if you can, use the `tfTMSF` value (Track, Minute, Second, Frame) for the `TimeFormat` property of the Media Player component to access positional properties (such as `Position` and `Length`). Extracting the values is not too complex if you use the proper functions of the MmSystem unit, such as the following:

```
CurrentTrack := Mci_TMSF_Track (MediaPlayer1.Position);
```

Here are the two functions that compute the values for the whole disk and for the current track:

```
procedure TForm1.CheckDisk;
var
  NTracks, NLen: Integer;
begin
  NTracks := MediaPlayer1.Tracks;
  NLen := MediaPlayer1.Length;
  DiskLabel.Caption := Format (
    'Tracks: %.2d, Length:%.2d:%.2d', [NTracks,
    Mci_TMSF_Minute (NLen), Mci_TMSF_Second (NLen)]);
  SpinEdit1.MaxValue := NTracks;
end;

procedure TForm1.CheckPosition;
var
  CurrentTrack, CurrentPos, TrackLen: Integer;
begin
  CurrentPos := MediaPlayer1.Position;
  CurPosLabel.Caption := Format ('Position: %.2d:%.2d',
    [Mci_TMSF_Minute (CurrentPos),
    Mci_TMSF_Second (CurrentPos)]);
  CurrentTrack := Mci_TMSF_Track (CurrentPos);
  TrackLen := MediaPlayer1.TrackLength [CurrentTrack];
  TrackNumberLabel.Caption := Format (
    'Current track: %.2d, Length:%.2d:%.2d', [CurrentTrack,
    Mci_MSF_Minute (TrackLen), Mci_MSF_Second (TrackLen)]);
end;
```

The code seems complex only because of the many conversions it makes. Notice in particular that the length of the current track (stored in the `TrackLength` property) is not measured using the default format, as the online help suggests, but with the `MSF` (Minute Second Frame) format.

The global values for the disk are computed only at startup and when the New CD button is clicked:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.TimeFormat := tfTMSF;
  MediaPlayer1.Open;
  CheckDisk;
  CheckPosition;
end;

procedure TForm1.NewButtonClick(Sender: TObject);
begin
  CheckDisk;
  CheckPosition;
end;
```

The values for the current track and position are computed this way each time the timer interval elapses, by calling the `CheckPosition` method. This is far from perfect, because if you want to play an audio CD while using other programs, a timer accessing the Media Player information often slows down the system too much. Of course, this mainly depends on your hardware. Besides telling the user what is going on, the form has the Media Player component to allow the user to start and stop playing, change tracks, and so on. The operations on this component activate and halt the timer:

```
procedure TForm1.MediaPlayer1PostClick(
  Sender: TObject; Button: TMPBtnType);
begin
  if MediaPlayer1.Mode = mpPlaying then
    Timer1.Enabled := True
  else
    Timer1.Enabled := False;
  CheckPosition;
end;
```

You can also use the Go button to jump to the track selected in the SpinEdit component, where the `MaxValue` property is set by the `CheckDisk` method. Here is the code I've written:

```
procedure TForm1.GoButtonClick(Sender: TObject);
var
  Playing: Boolean;
begin
  Playing := (MediaPlayer1.Mode = mpPlaying);
  if Playing then
    MediaPlayer1.Stop;
  MediaPlayer1.Position :=
    MediaPlayer1.TrackPosition[SpinEdit1.Value];
  CheckPosition;
  if Playing then
    MediaPlayer1.Play;
end;
```

A good extension to this program would be to connect it to a CD database with the title of each CD you own and the title of each track. (I would have done that if it hadn't been for the time it would have taken to enter the title and track of each of my disks.) Remember, anyway, that a similar program is already available in Windows.

What's Next

In this chapter, we have seen how to add some audio and video capabilities to Delphi applications. We have seen how to add sound effects to respond to user actions. We have also seen examples of how to use the Media Player with sound files, video files, and an external device (an audio CD). With computers and CD-ROM players becoming faster every year, video is becoming an important feature of many applications. Don't underestimate this area of programming simply because you are writing *serious* business programs.

EPILOGUE

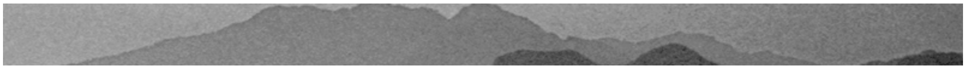
Delphi is a great programming environment. Now that you have learned about its core features, and seen the development of a number of programs, you might want to understand more about the language and fully master the internals of the VCL. This is where one of the books of the Mastering Delphi series I've written might help you. But you might also find more information in the other free e-books available on my web site.

As I mentioned in the Introduction, check the book page page, at

```
http://www.marcocantu.com/edelphi
```

for corrections and updates; the site has also links to other Delphi sites, documentation, simple wizards and components, and hosts also a newsgroup, which is the preferred way to report problems about this book (in the related section). It is possible to read the newsgroup also directly on the web from my site, and also sign-up to a mailing list in which I'll announce future edition of this and other volumes.

Again, as discussed in the introduction donations are welcome, also in the form of code examples, chapters for this and other books... but also offering cash, buying a printed book, or attending one of the Delphi classes I set up.



MASTERING™ DELPHI™ 6

Marco Cantù

SYBEX®

San Francisco • Paris • Düsseldorf • Soest • London

Associate Publisher: Richard Mills
Contracts and Licensing Manager: Kristine O'Callaghan
Acquisitions Editor: Denise Santoro Lincoln
Developmental Editors: Diane Lowery and Denise Santoro Lincoln
Editor: Pete Gaughan
Production Editor: Leslie E. H. Light
Technical Editors: Danny Thorpe and Eddie Churchill
Book Designer: Robin Kibby
Graphic Illustrator: Tony Jonick
Electronic Publishing Specialist: Kris Warrenburg, Cyan Design
Proofreaders: Nanette Duffy, Amey Garber, Jennifer Greiman,
Emily Hsuan, Laurie O'Connell, Nancy Riddiough
Indexer: Ted Laux
CD Coordinator: Christine Harris
CD Technician: Kevin Ly
Cover Designer: Design Site
Cover Illustrator/Photographer: Sergie Loobkoff

Library of Congress Card Number: 2001088115
ISBN: 0-7821-2874-2

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.
The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Software License Agreement: Terms and Conditions

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the "Software") to be used in connection with the book. SYBEX hereby grants to you a license to use the Software, subject to the terms that follow. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of SYBEX unless otherwise indicated and is protected by copyright to SYBEX or other copyright owner(s) as indicated in the media files (the "Owner(s)"). You are hereby granted a single-user license to use the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of SYBEX and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties ("End-User License"), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

Reusable Code in This Book

The author created reusable code in this publication expressly for reuse for readers. Sybex grants readers permission to reuse for any purpose the code found in this publication or its accompanying CD-ROM so long as the author is attributed in any application containing the reusable code, and the code itself is never sold or commercially exploited as a stand-alone product.

Software Support

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material but they are not supported by SYBEX. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate readme files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, SYBEX bears no responsibility. This notice concerning support for the Software is provided for your information only. SYBEX is not the agent or principal of the Owner(s), and SYBEX is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

Warranty

SYBEX warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from SYBEX in any other form or media than that enclosed herein or posted to www.sybex.com. If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

SYBEX Inc.
Customer Service Department
1151 Marina Village Parkway
Alameda, CA 94501
(510) 523-8233
Fax: (510) 523-2373
e-mail: info@sybex.com
WEB: HTTP://WWW.SYBEX.COM

After the 90-day period, you can obtain replacement media of identical format by sending us the defective disk, proof of purchase, and a check or money order for $10, payable to SYBEX.

Disclaimer

SYBEX makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will SYBEX, its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, SYBEX further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by SYBEX reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

Shareware Distribution

This Software may contain various programs that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

Copy Protection

The Software in whole or in part may or may not be copy-protected or encrypted. However, in all cases, reselling or redistributing these files without authorization is expressly forbidden except as specifically provided for by the Owner(s) therein.

To Lella, the love of my life,
and Benedetta, our love come to life.

Acknowledgments

This edition of *Mastering Delphi* marks the seventh year of the Delphi era, as it took Borland two years to release the latest incarnation of Delphi (along with its Linux twin, Kylix). As it has for many other programmers, Delphi has been my primary interest throughout these years; and writing, consulting, teaching, and speaking at conferences about Delphi have absorbed more and more of my time, leaving other languages and programming tools in the dust of my office. Because my work and my life are quite intertwined, many people have been involved in both, and I wish I had enough space and time to thank them all as they deserve. Instead, I'll just mention a few particular people and say a warm "Thank You" to the entire Delphi community (especially for the Spirit of Delphi 1999 Award I've been happy to share with Bob Swart).

The first official thanks are for the Borland programmers and managers who made Delphi possible and continue to improve it: Chuck Jazdzewski, Danny Thorpe, Eddie Churchill, Allen Bauer, Steve Todd, Mark Edington, Jim Tierney, Ravi Kumar, Jörg Weingarten, Anders Ohlsson, and all the others I have not had a chance to meet. I'd also like to give particular mention to my friends Ben Riga (the current Delphi product manager), John Kaster and David Intersimone (at Borland's Developer Relations), and others who have worked at Borland, including Charlie Calvert, Zack Urlocker and Nan Borreson.

The next thanks are for the Sybex editorial and production crew, many of whom I don't even know. Special thanks go to Pete Gaughan, Leslie Light, Denise Santoro Lincoln, and Diane Lowery; I'd also like to thank Richard Mills, Kristine O'Callaghan, and Kris Warrenburg.

This edition of *Mastering Delphi* has once again had an incredibly picky and detailed review from Delphi R&D team member Danny Thorpe. His highlights and comments in this and past editions have improved the book in all areas: technical content, accuracy, examples, and even readability. Thanks a lot. Previous editions also had special contributions: Tim Gooch worked on Part V for *Mastering Delphi 4*, and Giuseppe Madaffari contributed database material for the Delphi 5 edition. For this edition, Guy Smith-Ferrier rewrote the chapter on ADO, and Nando Dessena helped me with the InterBase chapter. Many improvements to the text and sample programs were suggested by technical reviewers of past editions (Juancarlo Añez, Ralph Friedman, Tim Gooch, and Alain Tadros) and in other reviews over the years by Bob Swart, Giuseppe Madaffari, and Steve Tendon.

Special thanks go to my friends Bruce Eckel, Andrea Provaglio, Norm McIntosh, Johanna and Phil of the BUG-UK, Ray Konopka, Mark Miller, Cary Jensen, Chris Frizelle of *The Delphi Magazine*, Foo Say How, John Howe, Mike Orriss, Chad "Kudzu" Hower, Dan Miser, Marco Miotti, and the entire D&D Team (Paolo, Andrea, Uberto, Nando, Giuseppe, and Mr. Coke). Also, a very big "Thank You" to all the attendees of my Delphi programming courses, seminars, and conferences in Italy, the United States, France, the United Kingdom, Singapore, the Netherlands, Germany, Sweden…

My biggest thanks go to my wife Lella who had to endure yet another many-months-long book-writing session and too many late nights (after spending the evenings with our daughter, Benedetta—I'll thank her with a hug, as Daddy's book looks quite boring to her). Many of our friends (and their kids) provided healthy breaks in the work: Sandro and Monica with Luca, Stefano and Elena, Marco and Laura with Matteo, Bianca, Luca and Elena with Tommaso, Chiara and Daniele with Leonardo, Laura, Vito and Marika with Sofia. Our parents, brothers, sisters, and their families were very supportive, too. It was nice to spend some of our free time with them and our six nephews—Matteo, Andrea, Giacomo, Stefano, Andrea, and Pietro.

Finally, I would like to thank all of the people, many of them unknown, who enjoy life and help to build a better world. If I never stop believing in the future and in peace, it is also because of them.

Introduction

The first time Zack Urlocker showed me a yet-to-be-released product code-named Delphi, I realized that it would change my work—and the work of many other software developers. I used to struggle with C++ libraries for Windows, and Delphi was and still is the best combination of object-oriented programming and visual programming for Windows.

Delphi 6 simply builds on this tradition and on the solid foundations of the VCL to deliver another astonishing and all-encompassing software development tool. Looking for database, client/server, multitier, intranet, or Internet solutions? Looking for control and power? Looking for fast productivity? With Delphi 6 and the plethora of techniques and tips presented in this book, you'll be able to accomplish all this.

Six Versions and Counting

Some of the original Delphi features that attracted me were its form-based and object-oriented approach, its extremely fast compiler, its great database support, its close integration with Windows programming, and its component technology. But the most important element was the Object Pascal language, which is the foundation of everything else.

Delphi 2 was even better! Among its most important additions were these: the Multi-Record Object and the improved database grid, OLE Automation support and the variant data type, full Windows 95 support and integration, the long string data type, and Visual Form Inheritance. Delphi 3 added to this the code insight technology, DLL debugging support, component templates, the TeeChart, the Decision Cube, the WebBroker technology, component packages, ActiveForms, and an astonishing integration with COM, thanks to interfaces.

Delphi 4 gave us the AppBrowser editor, new Windows 98 features, improved OLE and COM support, extended database components, and many additions to the core VCL classes, including support for docking, constraining, and anchoring controls. Delphi 5 added to the picture many more improvements of the IDE (too many to list here), extended database support (with specific ADO and InterBase datasets), an improved version of MIDAS with Internet support, the TeamSource version-control tool, translation capabilities, the concept of frames, and new components.

Now Delphi 6 adds to all these features support for cross-platform development with the new Component Library for Cross-Platform (CLX), an extended run-time library, the new dbExpress database engine, Web services and exceptional XML support, a powerful Web development framework, more IDE enhancements, and a plethora of new components and classes, as you'll see in the following pages.

Delphi is a great tool, but it is also a complex programming environment that involves many elements. This book will help you master Delphi programming, including the Object Pascal language, Delphi components (both using the existing ones and developing your own), database and client/server support, the key elements of Windows and COM programming, and Internet and Web development.

You do not need in-depth knowledge of any of these topics to read this book, but you do need to know the basics of Pascal programming. Having some familiarity with Delphi will help you considerably, particularly after the introductory chapters. The book starts covering its topics in depth immediately; much of the introductory material from previous editions has been removed. Some of this material and an introduction to Pascal is available on the companion CD-ROM and on my Web site and can be a starting point if you are not confident with Delphi basics. Each new Delphi 6 feature is covered in the relevant chapters throughout the book.

The Structure of the Book

The book is divided into four parts:

- Part I, "Foundations," introduces new features of the Delphi 6 Integrated Development Environment (IDE) in Chapter 1, then moves to the Object Pascal language and to the run-time library (RTL) and Visual Component Library (VCL), providing both foundations and advanced tips.
- Part II, "Visual Programming," covers standard components, Windows common controls, graphics, menus, dialogs, scrolling, docking, multipage controls, Multiple Document Interface, the Action List and Action Manager architectures, and many other topics. The focus is on both the VCL and CLX libraries. The final chapters discuss the development of custom components and the use of libraries and packages.
- Part III, "Database Programming," covers plain database access, in-depth coverage of the data-aware controls, client/server programming, dbExpress, InterBase, ADO and dbGo, DataSnap (or MIDAS), and the development of custom data-aware controls and data sets.

- Part IV, “Beyond Delphi: Connecting with the World,” first discusses COM, OLE Automation, and COM+. Then it moves to Internet programming, covering TCP/IP sockets, Internet protocols and Indy, Web server-side extensions (with WebBroker and WebSnap), XML, and the development of Web services.

As this brief summary suggests, the book covers topics of interest to Delphi users at nearly all levels of programming expertise, from “advanced beginners” to component developers.

In this book, I’ve tried to skip reference material almost completely and focus instead on techniques for using Delphi effectively. Because Delphi provides extensive online documentation, to include lists of methods and properties of components in the book would not only be superfluous, it would also make it obsolete as soon as the software changes slightly. I suggest that you read this book with the Delphi Help files at hand, to have reference material readily available.

However, I’ve done my best to allow you to read the book away from a computer if you prefer. Screen images and the key portions of the listings should help in this direction. The book uses just a few conventions to make it more readable. All the source code elements, such as keywords, properties, classes, and functions, appear in `this font`, and code excerpts are formatted as they appear in the Delphi editor, with boldfaced keywords and italic comments and strings.

Free Source Code on CD (and the Web)

This book focuses on examples. After the presentation of each concept or Delphi component, you’ll find a working program example (sometimes more than one) that demonstrates how the feature can be used. All told, there are about 300 examples presented in the book. These programs are directly available on the companion CD-ROM. The same material is also available on my Web site (`www.marcocantu.com`), where you’ll also find updates and examples from past editions. Inside the back cover of the book, you’ll find more information about the CD. Most of the examples are quite simple and focus on a single feature. More complex examples are often built step-by-step, with intermediate steps including partial solutions and incremental improvements.

NOTE Some of the database examples also require you to have the Delphi sample database DBDEMOS installed; it is part of the default Delphi installation. Others require the InterBase EMPLOYEE sample database.

Beside the source code files, the CD hosts the ready-to-use compiled programs. There is also an HTML version of the source code, with full syntax highlighting, along with a complete cross-reference of keywords and identifiers (class, function, method, and property names, among others). The cross-reference is an HTML file, so you'll be able to use your browser to easily find all the programs that use a Delphi keyword or identifier you're looking for (not a full search engine, but close enough).

The directory structure of the sample code is quite simple. Basically, each chapter of the book has its own folder, with a subfolder for each example (e.g., `06\Borders`). In the text, the examples are simply referenced by name (e.g., Borders).

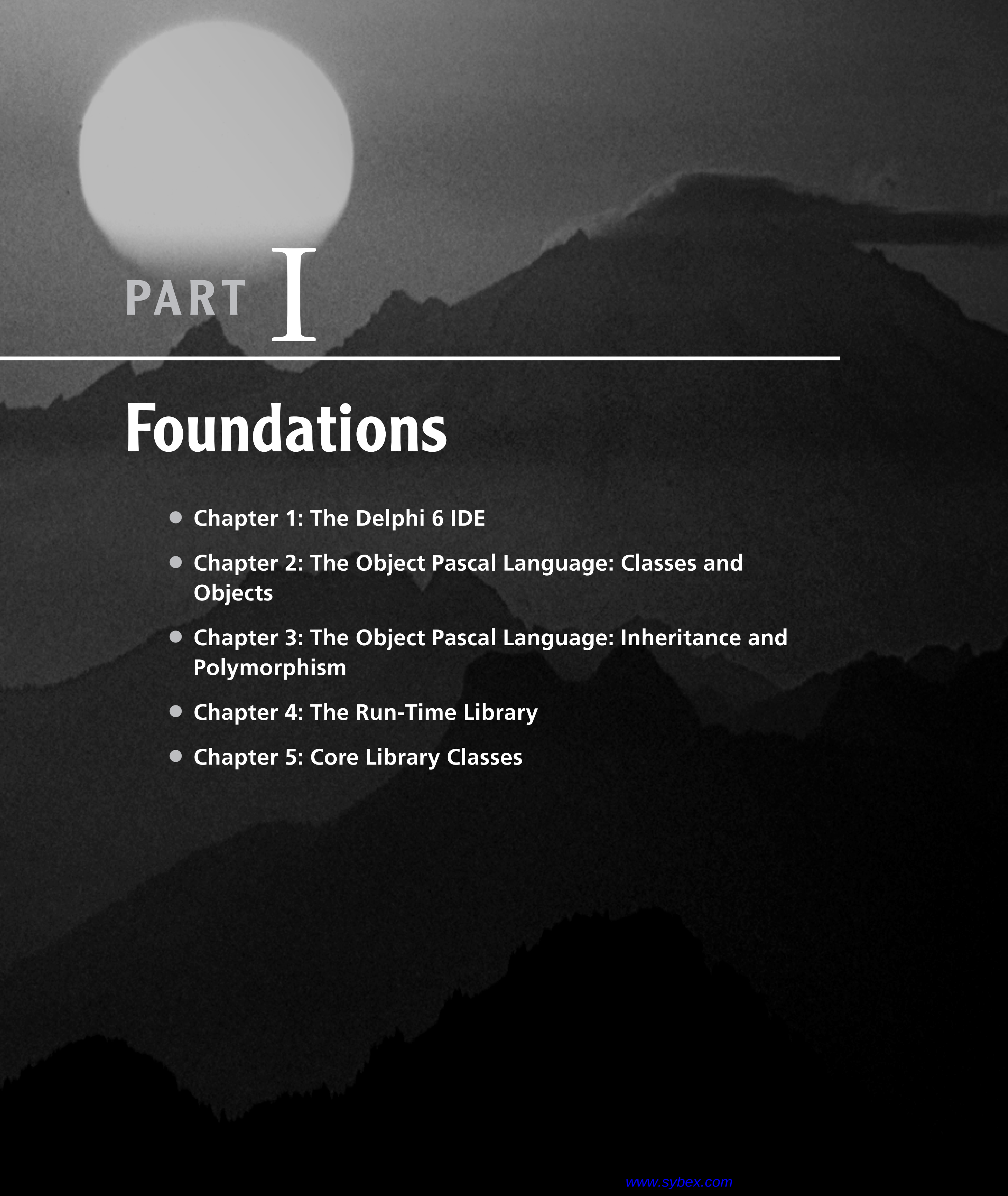
TIP To change an example, first copy it (or the entire `md6code` folder) to your hard disk, but before opening it remember to set the read-only flag to False (it is True by default on the read-only media)

NOTE Be sure to read the source code archive's Readme file, which contains important information about using the software legally and effectively.

How to Reach the Author

If you find any problems in the text or examples in this book, both the publisher and I would be happy to hear from you. Besides reporting errors and problems, please give us your unbiased opinion of the book and tell us which examples you found most useful and which you liked least. There are several ways you can provide this feedback:

- On the Sybex Web site (`www.sybex.com`), you'll find updates to the text or code as necessary. To comment on this book, click the Contact Sybex link and then choose Book Content Issues. This link displays a form where you can enter your comments.
- My own Web site (`www.marcocantu.com`) hosts further information about the book and about Delphi, where you might find answers to your questions. The site has news and tips, technical articles, free online books, white papers, Delphi links, and my collection of Delphi components and tools.
- I have also set up a newsgroup section specifically devoted to my books and to general Delphi Q&A. Refer to my Web site for a list of the newsgroup areas and for the instructions to subscribe to them. (In fact, these newsgroups are totally free but require a login password.) The newsgroups can also be accessed via a Web interface you can find on my site.
- Finally, you can reach me via e-mail at `marco@marcocantu.com`. For technical questions, please try using the newsgroups first, as you might get answers earlier and from multiple people. My mailbox is usually quite full and, regretfully, I cannot reply promptly to every request. (Please write to me in English or Italian.)



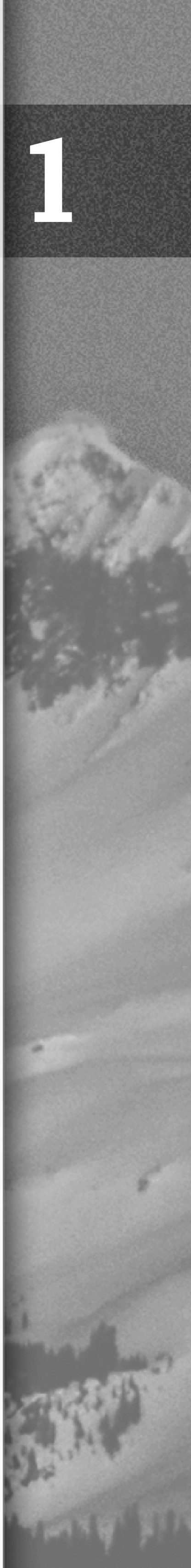
PART I

Foundations

- Chapter 1: The Delphi 6 IDE
- Chapter 2: The Object Pascal Language: Classes and Objects
- Chapter 3: The Object Pascal Language: Inheritance and Polymorphism
- Chapter 4: The Run-Time Library
- Chapter 5: Core Library Classes

CHAPTER 1

The Delphi 6 IDE



- Object TreeView and Designer view
- The AppBrowser editor
- The code insight technology
- Designing forms
- The Project Manager
- Delphi files

In a visual programming tool such as Delphi, the role of the environment is at times even more important than the programming language. Delphi 6 provides many new features in its visual development environment, and this chapter covers them in detail. This chapter isn't a complete tutorial but mainly a collection of tips and suggestions aimed at the average Delphi user. In other words, it's not for newcomers. I'll be covering the new features of the Delphi 6 Integrated Development Environment (IDE) and some of the advanced and little-known features of previous versions as well, but in this chapter I won't provide a step-by-step introduction. Throughout this book, I'll assume you already know how to carry out the basic hands-on operations of the IDE, and all the chapters after this one focus on programming issues and techniques.

If you *are* a beginning programmer, don't be afraid. The Delphi Integrated Development Environment is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi CD) with a tutorial that introduces the development of Delphi applications. You can also find a step-by-step introduction to the Delphi IDE on my Web site, **`http://www.marcocantu.com`**. The short online book *Essential Delphi* is based on material from the first chapters of earlier editions of *Mastering Delphi*.

Editions of Delphi 6

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

- The "Personal" edition is aimed at Delphi newcomers and casual programmers and has support for neither database programming nor any of the other advanced features of Delphi 6.
- The "Professional" edition is aimed at professional developers. It includes all the basic features, plus database programming support (including ADO support), basic Web server support (WebBroker), and some of the external tools. This book generally assumes you are working with at least the Professional edition.
- The "Enterprise" edition is aimed at developers building enterprise applications. It includes all the new XML and advanced Web services technologies, internationalization, three-tier architecture, and many other tools. Some chapters of this book cover features included only in Delphi Enterprise; these sections are specifically identified.

NOTE In the past, some of the features of Delphi Enterprise have been available as an "up-sell" to owners of Delphi Professional. This might also happen for this version.

Besides the different editions available, there are ways to customize the Delphi environment. In the screen illustrations throughout the book, I've tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which might be reflected in some of the screen shots.

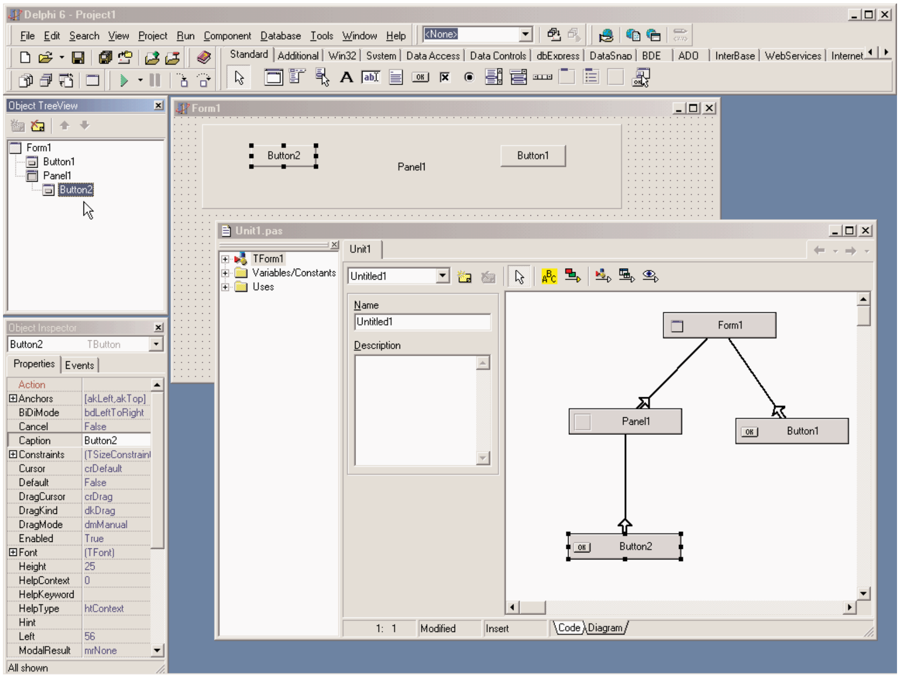
The Delphi 6 IDE

The Delphi 6 IDE includes large and small changes that will really improve a programmer's productivity. Among the key features are the introduction of the Object TreeView for every designer, an improved Object Inspector, extended code completion, and loadable views, including diagrams and HTML.

Most of the features are quite easy to grasp, but it's worth examining them with some care so that you can start using Delphi 6 at its full potential right away. You can see an overall image of Delphi 6 IDE, highlighting some of the new features, in Figure 1.1.

FIGURE 1.1:

The Delphi 6 IDE: Notice the Object TreeView and the Diagram view.



Name: Untitled1

Description:

Object Inspector

Button2 TButton

Properties | Events

Action	
⊞Anchors	[akLeft,akTop]
BiDiMode	bdLeftToRight
Cancel	False
Caption	Button2
⊞Constraints	(TSizeConstrain
Cursor	crDefault
Default	False
DragCursor	crDrag
DragKind	dkDrag
DragMode	dmManual
Enabled	True
⊞Font	(TFont)
Height	25
HelpContext	0
HelpKeyword	
HelpType	htContext
Hint	
Left	56
ModalResult	mrNone

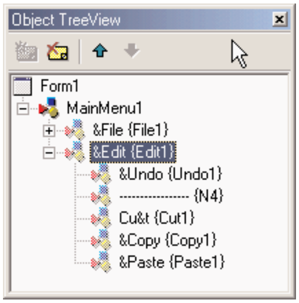
All shown

The Object TreeView

Delphi 5 introduced a TreeView for data modules, where you could see the relations among nonvisual components, such as datasets, fields, actions, and so on. Delphi 6 extends the idea by providing an Object TreeView for every designer, including plain forms. The Object TreeView is placed by default above the Object Inspector; use the View ➢ Object TreeView command in case it is hidden.

The Object TreeView shows all of the components and objects on the form in a tree, representing their relations. The most obvious is the parent/child relation: Place a panel on a form, a button inside it and one outside of the panel. The tree will show the two buttons, one under the form and the other under the panel, as in Figure 1.1. Notice that the TreeView is synchronized with the Object Inspector and Form Designer, so as you select an item and change the focus in any one of these three tools, the focus changes in the other two tools.

Besides parent/child, the Object TreeView shows also other relations, such as owner/owned, component/subobject, collection/item, plus various specific ones, including dataset/connection and data source/dataset relations. Here, you can see an example of the structure of a menu in the tree.



```
Object TreeView
Form1
  MainMenu1
    &File {File1}
    &Edit {Edit1}
      &Undo {Undo1}
      --------------- {N4}
      Cu&t {Cut1}
      &Copy {Copy1}
      &Paste {Paste1}
```

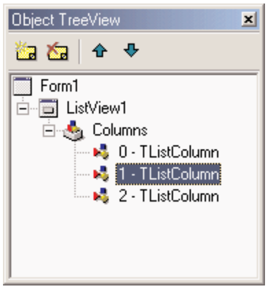
At times, the TreeView also displays “dummy” nodes, which do not correspond to an actual object but do correspond to a predefined one. As an example of this behavior, drop a Table component (from the BDE page) and you’ll see two grayed icons for the session and the alias. Technically, the Object TreeView uses gray icons for components that do not have design-time persistence. They are real components (at design time and at run time), but because they are default objects that are constructed at run time and have no persistent data that can be edited at design time, the Data Module Designer does not allow you to edit their properties. If you drop a Table on the form, you’ll also see items with a red question mark enclosed in a yellow circle next to them. This symbol indicates partially undefined items (there used to be a red square around those items in Delphi 5).

The Object TreeView supports multiple types of *dragging*:

- You can select a component from the palette (by clicking it, not actually dragging it), move the mouse over the tree, and click a component to drop it there. This allows you to drop a component in the proper container (form, panel, and others) regardless of the fact that its surface might be totally covered by other components, something that prevents you from dropping the component in the designer without first rearranging those components.
- You can drag components within the TreeView—for example, moving a component from one container to another—something that, with the Form Designer, you can do only with cut and paste techniques. Moving instead of cutting provides the advantage that if you have connections among components, these are not lost, as happens when you delete the component during the cut operation.
- You can drag components from the TreeView to the Diagram view, as we'll see later.

Right-clicking any element of the TreeView displays a shortcut menu similar to the component menu you get when the component is in a form (and in both cases, the shortcut menu may include items related to the custom component editors). You can even delete items from the tree.

The TreeView doubles also as a collection editor, as you can see here for the `Columns` property of a ListView control. In this case, you can not only rearrange and delete items, but also add new items to the collection.



TIP You can print the contents of the Object TreeView for documentation purposes. Simply select the window and use the File ➢ Print command, as there is no Print command in the shortcut menu.

Loadable Views

Another important change has taken place in the Code Editor window. For any single file loaded in the IDE, the editor can now show multiple views, and these views can be defined programmatically and added to the system, then loaded for given files—hence the name *loadable* views.

The most frequently used view is the Diagram page, which was already available in Delphi 5 data modules, although it was less powerful. Another set of views is available in Web applications, including an HTML Script view, an HTML Result preview, and many others discussed in Chapter 22.

The Diagram View

Along with the TreeView, another feature originally introduced in Delphi 5 Data Modules and now available for every designer is the Diagram view. This view shows dependencies among components, including parent/child relations, ownership, linked properties, and generic relations. For dataset components, it also supports master/detail relations and lookup connections. You can even add your comments in text blocks linked to specific components.

The Diagram is not built automatically. You must drag components from the TreeView to the diagram, which will automatically display the existing relations among the components you drop there. In Delphi 6, you can now select multiple items from the Object TreeView and drag them all at once to the Diagram page.

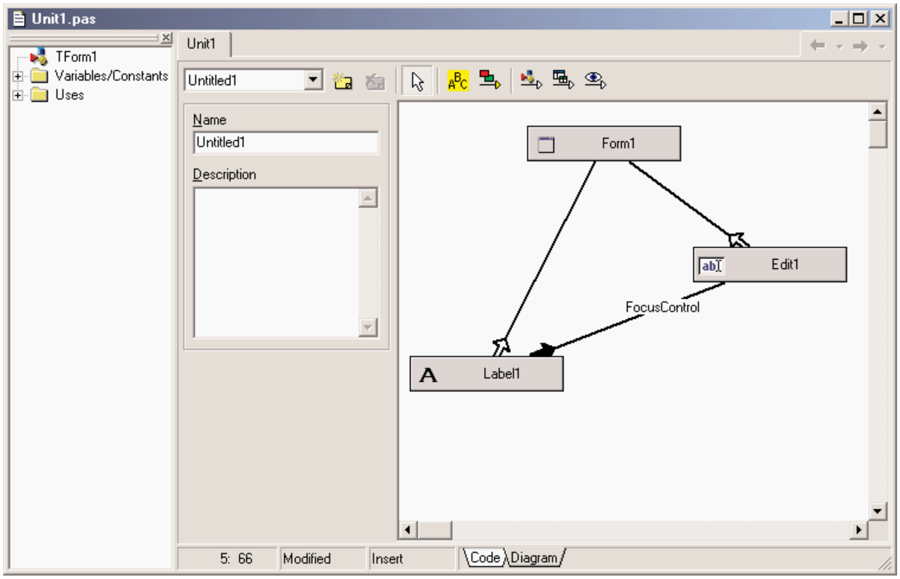
What's nice is that you can set properties by simply drawing arrows between the components. For example, after moving an edit and a label to Diagram, you can select the Property Connector icon, click the label, and drag the mouse cursor over the edit. When you release the mouse button, the Diagram will set up a property relation based on the `FocusControl` property, which is the only property of the label referring to an edit control. This situation is depicted in Figure 1.2.

As you can see, setting properties is *directional*: If you drag the property relation line from the edit to the label, you end up trying to use the label as the value of a property of the edit box. Because this isn't possible, you'll see an error message indicating the problem and offering to connect the components in the opposite way.

In Delphi 6, the Diagram view allows you to create multiple diagrams for each Delphi unit—that is, for each form or data module. Simply give a name to the diagram and possibly add a description, click the New Diagram button, prepare another diagram, and you'll be able to switch back and forth between diagrams using the combo box available in the toolbar of the Diagram view.

FIGURE 1.2:

The Diagram view allows you to connect components using the Property connector.



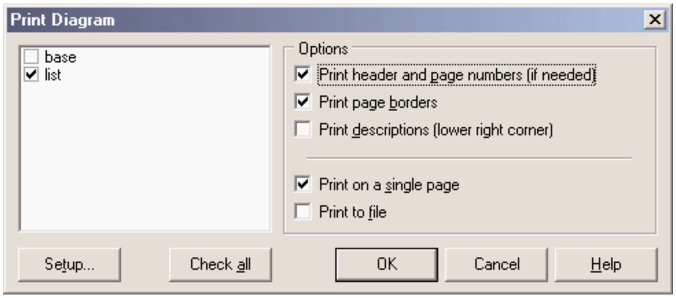
Name: Untitled1

Description:

Although you can use the Diagram view to set up relations, its main role is to document your design. For this reason, it is important to be able to print the content of this view. Using the standard File ➢ Print command while the Diagram is active, Delphi prompts you for options, as you can see in Figure 1.3, allowing you to customize the output in many ways.

FIGURE 1.3:

The Print Options for the Diagram view



Print Diagram

- □ base
- ☑ list

Options

- ☑ Print header and page numbers (if needed)
- ☑ Print page borders
- □ Print descriptions (lower right corner)
- ☑ Print on a single page
- □ Print to file

The information in the Data Diagram view is saved in a separate file, not as part of the DFM file. Delphi 5 used design-time information (DTI) files, which had a structure similar to INI files. Delphi 6 can still read the older .DTI format, but uses the new Delphi Diagram Portfolio format (.DDP). These files apparently use the DFM binary format (or a similar one), so they are not editable as text. All of these files are obviously useless at run time (it makes no sense to include them in the compilation of the executable file).

An IDE for Two Libraries

Another very important change I just want to introduce here is the fact that Delphi 6, for the first time, allows you to use to different component libraries, VCL (Visual Components Library) and CLX (Component Library for Cross-Platform). When you create a new project, you simply choose which of the two libraries you want to use, starting with the File ➢ New ➢ Application command for a classic VCL-based Windows program and with the File ➢ New ➢ CLX Application command for a new CLX-based portable application.

Creating a new project or opening an existing one, the Component Palette is rearranged to show only the controls related to the current library (although most of them are actually shared). This topic is fully covered in Chapter 6, so I don't want to get into the details here; I'll just underline that you can use Delphi 6 to build applications you can compile right away for Linux using Kylix. The effect of this change on the IDE is really quite large, as many things "under the hood" had to be reengineered. Only programmers using the ToolsAPI and other advanced elements will notice all these internal differences, as they are mostly transparent to most users.

Smaller Enhancements

Besides this important change and others I'll discuss in later sections, such as the update of the Object Inspector and of code completion, there are small (but still quite important) changes in the Delphi 6 IDE. Here is a list of these changes:

- There is a new Window menu in the IDE. This menu lists the open windows, something you could obtain in the past using the Alt+0 keys. This is really very handy, as windows often end up behind others and are hard to find. (Thanks, Borland, for listening to this and other simple but effective requests from users.)

TIP Two entries of the Main Window registry section of Delphi (under `\Software\Borland\Delphi\6.0` for the current user) allow you to hide this menu and disable its alphabetic sort order. This registry keys use strings (in place of Boolean values) where "-1" indicates true and "0" false.

- The File menu doesn't include specific items for creating new forms or applications. These commands have been increased in number and grouped under the File ➢ New secondary menu. The Other command of this menu opens the New Item dialog box (the Object Repository) as the File ➢ New command did in the past.
- The Component Palette local menu has a submenu listing all of the palette pages in alphabetic order. You can use it to change the active page, particularly when it is not visible on the screen.

TIP

The order of the entries in the Tabs submenu of the Component Palette local menu can be set in the same order as the palette itself, and not sorted alphabetically. This is accomplished by setting to "0" (false) the value of the Sort Palette Tabs Menu key of the Main Window registry section of Delphi (under `\Software\Borland\Delphi\6.0` for the current user).

- There is a new toolbar, the Internet toolbar, which is initially disabled. This toolbar supports WebSnap applications.

Updated Environment Options Dialog Box

Quite a few small changes relate to the commonly used Environment Options dialog box. The pages of this dialog box have been rearranged, moving the Form Designer options from the Preferences page to the new Designer page. There are also a few new options and pages:

- The Preferences page of the Environment Options dialog box has a new check box that prevents Delphi windows from automatically docking with each other. This is a very welcome addition!
- A new page, Environment Variables, allows you to see system environment variables (such as the standard path names and OS settings) and set user-defined variables. The nice point is that you can use both system- and user-defined environment variables in each of the dialog boxes of the IDE—for example, you can avoid hard-coding commonly used path names, replacing them with a variable. In other words, the environment variables work similarly to the $DELPHI variable, referring to Delphi's base directory, but can be defined by the user.
- Another new page is called Internet. In this page, you can choose the default file extensions used for HTML and XML files (mainly by the WebSnap framework) and also associate an external editor with each extension.

Delphi Extreme Toys

At times, the Delphi team comes up with small enhancements of the IDE that aren't included in the product because they either aren't of general use or will require time to be improved in quality, user interface, or robustness. Some of these internal wizards and IDE extensions have now been made available, with the collective name of Delphi Extreme Toys, to registered Delphi 6 users. You should automatically get this add-on as you register your copy of the product (online or through a Borland office).

There isn't an official list of the content of the Extreme Toys, as Borland plans to keep extending them. The initial release includes an IDE-based search engine for seeking answers on Delphi across the Internet, a wizard for turning on and off specific compiler warnings, and an “invokamatic” wizard for accelerating the creation of Web services. The Extreme Toys will, in essence, be *unofficial* wizards, code utilities, and components from the Delphi team—or useful stuff from various people.

Recent IDE Additions

Delphi 5 provided a huge number of new features to the IDE. In case you’ve only used versions of Delphi prior to 5, or need to brush up on some useful added information, this is a short summary of the most important of the features introduced in Delphi 5.

Saving the Desktop Settings

The Delphi IDE allows programmers to customize it in various ways—typically, opening many windows, arranging them, and docking them to each other. However, programmers often need to open one set of windows at design time and a different set at debug time. Similarly, programmers might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

For this reason, Delphi allows you to save a given arrangement of IDE windows (called a *desktop*) with a name and restore it easily. Also, you can make one of these groupings your default debugging setting, so that it will be restored automatically when you start the debugger. All these features are available in the Desktops toolbar. You can also work with desktop settings using the View ➢ Desktops menu.

Desktop setting information is saved in DST files, which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its new property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78
```

```
[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1

[AlignmentPalette]
Create=1
Visible=0
...
```

Desktop settings override project settings. This helps eliminate the problem of moving a project between machines (or between developers) and having to rearrange the windows to your liking. Delphi 5 separates per-user and per-machine preferences from the project settings, to better support team development.

TIP

If you open Delphi and cannot see the form or other windows, I suggest you try checking (or deleting) the desktop settings. If the project desktop was last saved on a system running in a high-resolution video mode (or a multimonitor configuration) and opened on a different system with lower screen resolution or fewer monitors, some of the windows in the project might be located off-screen on the lower-resolution system. The simplest ways to fix that are either to load your own named desktop configuration after opening the project, thus overriding the project desktop settings, or just delete the DST file that came with the project files.

The To-Do List

Another feature added in Delphi 5 was the to-do list. This is a list of tasks you still have to do to complete a project, a collection of notes for the programmer (or programmers, as this tool can be very handy in a team). While the idea is not new, the key concept of the to-do list in Delphi is that it works as a two-way tool.

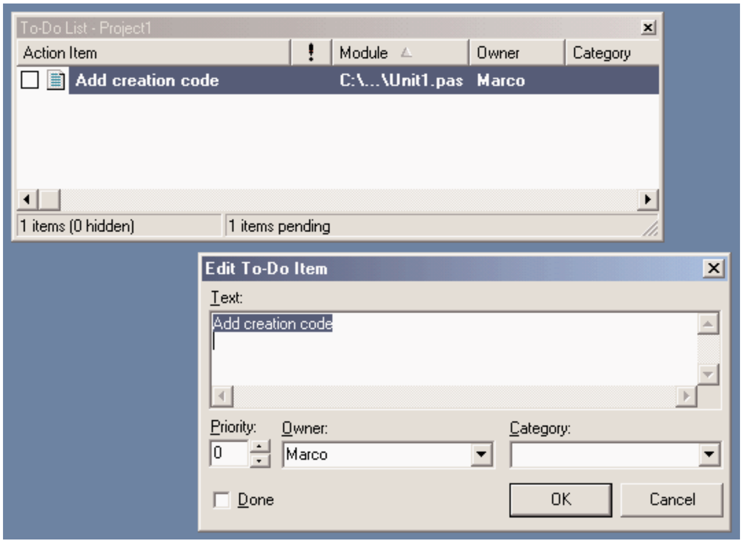
In fact, you can add or modify to-do items by adding special TODO comments to the source code of any file of a project; you'll then see the corresponding entries in the list. But you can also visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look like in the source code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // TODO -oMarco: Add creation code
end;
```

The same item can be visually edited in the window shown in Figure 1.4.

FIGURE 1.4:

The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code.



To-Do List - Project1

	Action Item	!	Module	Owner	Category
□	Add creation code		C:\...\Unit1.pas	Marco	

1 items (0 hidden) 1 items pending

Edit To-Do Item

Text: Add creation code

Priority: 0

Owner: Marco

Category:

□ Done

The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the Ctrl+A key combination in the To-Do List window or right-click in the window and select Add from the shortcut menu. These items are saved in a special file with the .TODO extension.

You can use multiple options with a `TODO` comment. You can use `–o` (as in the code excerpt above) to indicate the owner, the programmer who entered the comment; the `–c` option to indicate a category; or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

Delphi treats everything after the colon, up to the end of line or the closing brace, depending on the type of comment, as the text of the to-do item. Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from `TODO` to `DONE`. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, which can automatically collect to-do information from the source code files as you type them, sort and filter them, and export them to the Clipboard as plain text or an HTML table.

The AppBrowser Editor

The editor included with Delphi hasn't changed recently, but it has many features that many Delphi programmers don't know and use. It's worth briefly examining this tool. The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor, and you can also open multiple editor windows. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Shift+Ctrl+Tab to move in the opposite direction).

TIP In Delphi 6, you can drag-and-drop the tabs with the unit names in the upper portion of the editor to change their order, so that you can use a single Ctrl+Tab to move between the units you are mostly interested in. The local menu of the editor has also a Pages command, which lists all of the available pages in a submenu, a handy feature when many units are loaded.

Several options affect the editor, located in the new Editor Properties dialog box. You have to go to the Preferences page of the Environment Options dialog box, however, to set the editor's AutoSave feature, which saves the source code files each time you run the program (preventing data loss in case the program crashes badly).

I won't discuss the various settings of the editor, as they are quite intuitive and are described in the online Help. A tip to remember is that using the Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. You can also copy text instead of moving it, by pressing the Ctrl key while dragging.

The Code Explorer

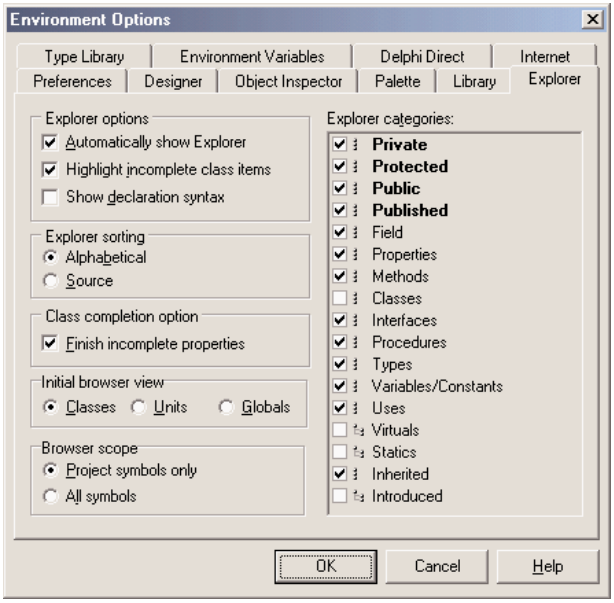
The Code Explorer window, which is generally docked on the side of the editor, simply lists all of the types, variables, and routines defined in a unit, plus other units appearing in `uses` statements. For complex types, such as classes, the Code Explorer can list detailed information, including a list of fields, properties, and methods. All the information is updated as soon as you start typing in the editor. You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration.

TIP In Delphi 6 you can modify variables, properties, and method names directly in the Code Explorer.

While all that is quite obvious after you've used Delphi for a few minutes, some features of the Code Explorer are not so intuitive. One important point is that you have full control of the layout of the information, and you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer. Collapsing the tree can help you make your selections more quickly. You can configure the Code Explorer by using the corresponding page of the Environment Options, as shown in Figure 1.5.

FIGURE 1.5:

You can configure the Code Explorer in the Environment Options dialog box.



Explorer options
☑ Automatically show Explorer
☑ Highlight incomplete class items
□ Show declaration syntax

Explorer sorting
● Alphabetical
○ Source

Class completion option
☑ Finish incomplete properties

Initial browser view
● Classes ○ Units ○ Globals

Browser scope
● Project symbols only
○ All symbols

Explorer categories:
☑ **Private**
☑ **Protected**
☑ **Public**
☑ **Published**
☑ Field
☑ Properties
☑ Methods
□ Classes
☑ Interfaces
☑ Procedures
☑ Types
☑ Variables/Constants
☑ Uses
□ Virtuals
□ Statics
☑ Inherited
□ Introduced

Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view—it simply adds the node in the tree. For example, if you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer. On the contrary, the used units are listed as main nodes instead of being kept in the `Uses` folder. I generally disable the Types, Classes, and Variables selections.

Because each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, as this requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer, in fact, provides a very handy way of navigating the source code of a large unit. When you double-click a method in the Code Explorer, the focus moves to the definition in the class declaration (in the interface portion of the unit). You can use the Ctrl+Shift combination with the Up or Down arrow keys to jump from the definition of a method or procedure in the interface portion of a unit to its complete definition in the implementation portion (or back again).

NOTE Some of the Explorer Categories shown in Figure 1.5 are used by the Project Explorer, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.

Browsing in the Editor

Another feature of the AppBrowser editor is *Tooltip symbol insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the Visual Component Library (VCL).

WARNING Although it may seem a good idea at first, you cannot use Tooltip symbol insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, in fact, the Tooltip won't appear.

The real bonus of this feature, however, is that you can turn it into a navigational aid. When you hold down the Ctrl key and move the mouse over the identifier, Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of Web browsers, and the pointer changes to a hand whenever it's positioned on the link.

For example, you can Ctrl+click the `TLabel` identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again as in a Web browser. You can also click the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

How can you jump directly to the VCL source code if it is not part of your project? The AppBrowser editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment.

Class Completion

The third important feature of Delphi's AppBrowser editor is *class completion*, activated by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, as Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi also simplify life in a similar way for programmers who write a little extra code behind event handlers. The new code-generation feature, in fact, applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration:

```
public
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }
procedure TForm1.Hello(MessageText: string);
begin
end;
```

This is really handy, compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin...end` code for every method copied.

Class completion can also work the other way around. You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

Code Insight

Besides the Code Explorer, class completion, and the navigational features, the Delphi editor supports the *code insight* technology. Collectively, the code insight techniques are based on a constant background parsing, both of the source code you write and of the source code of the system units your source code refers to.

Code insight comprises five capabilities: code completion, code templates, code parameters, Tooltip expression evaluation, and Tooltip symbol insight. This last feature was already covered in the section "Browsing in the Editor"; the other four will be discussed in the following subsections. You can enable, disable, and configure each of these features in the Code Insight page of the Editor Options dialog box.

Code Completion

Code completion allows you to choose the property or method of an object simply by looking it up on a list or by typing its initial letters. To activate this list, you just type the name of an object, such as `Button1`, then add the dot, and wait. To force the display of the list, press Ctrl+spacebar; to remove it when you don't want it, press Esc. Code completion also lets you look for a proper value in an assignment statement.

In Delphi 6, as you start typing, the list filters its content according to the initial portion of the element you've inserted. The code completion list uses colors and shows more details to help you distinguish different items. Another new feature is that in the case of functions with parameters, parentheses are included in the generated code, and the parameters list hint is displayed immediately.

As you type `:=` after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click it to change the order of the items, sorting either by scope or by name, and you can also resize the window.

In Delphi 6, code completion also works in the interface section of a unit. If you press Ctrl+spacebar while the cursor is inside the class definition, you'll get a list of: virtual methods you can override (including abstract methods), the methods of implemented interfaces, the base class properties, and eventually system messages you can handle. Simply selecting one of them will add the proper method to the class declaration. In this particular case, the code completion list allows multiple selection.

TIP

When the code you've written is incorrect, code insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific code insight errors in the Message pane (which must already be open; it doesn't open automatically to display compilation errors). To activate this feature, you need to set an undocumented registry entry, setting the string key `\Delphi\6.0\Compiling\ShowCodeInsiteErrors` to the value '1'.

There are advanced features of Delphi 6 code completion that aren't easy to spot. One that I found particularly useful relates to the discovery of symbols in units not used by your project. As you invoke it (with Ctrl+spacebar) over a blank line, the list also includes symbols from common units (such as Math, StrUtils, and DateUtils) not already included in the `uses` statement of the current one. By selecting one of these *external* symbols, Delphi adds the unit to the `uses` statement for you. This feature (which doesn't work inside expressions) is driven by a customizable list of extra units, stored in the registry key `\Delphi\6.0\CodeCompletion\ExtraUnits`.

Code Templates

Code templates allow you to insert one of the predefined code templates, such as a complex statement with an inner begin...end block. Code templates must be activated manually, by typing Ctrl+J to show a list of all of the templates. If you type a few letters (such as a keyword) before pressing Ctrl+J, Delphi will list only the templates starting with those letters.

You can add your own custom code templates, so that you can build your own shortcuts for commonly used blocks of code. For example, if you use the MessageDlg function often, you might want to add a template for it. In the Code Insight page of the Environment Options dialog box, click the Add button in the Code Template area, type in a new template name (for example, **mess**), type a description, and then add the following text to the template body in the Code memo control:

```
MessageDlg ('|', mtInformation, [mbOK], 0);
```

Now every time you need to create a message dialog box, you simply type **mess** and then press Ctrl+J, and you get the full text. The vertical line (or pipe) character indicates the position within the source code where the cursor will be in the editor after expanding the template. You should choose the position where you want to start typing to complete the code generated by the template.

Although code templates might seem at first sight to correspond to language keywords, they are in fact a more general mechanism. They are saved in the DELPHI32.DCI file, so it should be possible to copy this file to make your templates available on different machines. Merging two code template files is not documented, though.

Code Parameters

Code parameters display, in a hint or Tooltip window, the data type of a function's or method's parameters while you are typing it. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a pop-up hint window. To force the display of code parameters, you can press Ctrl+Shift+spacebar. As a further help, the current parameter appears in bold type.

Tooltip Expression Evaluation

Tooltip expression evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor.

More Editor Shortcut Keys

The editor has many more shortcut keys that depend on the editor style you've selected. Here are a few of the less-known shortcuts, most of which are useful:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a "gutter" margin on the side of the editor. To jump back to the bookmark, press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the facts that a new bookmark can override an existing one and that bookmarks are not persistent; they are lost when you close the file.
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is the one set by the Block Indent option in the General page of the Editor Properties dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R starts recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that might have little meaning once you've moved on to a different source code file. Editor macros are quite useful for performing multistep operations over and over again, such as reformatting source code or arranging data more legibly in source code.
- Holding down the Alt key, you can drag the mouse to select rectangular areas of the editor, not just consecutive lines and words.

The Form Designer

Another Delphi window you'll interact with very often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, you can select a component directly with the mouse or through the Object Inspector, a handy feature when a control is behind another one or is very small. If one control covers another completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click the selected component. This will deselect the current component and select the form by default.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the Left and Top properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position. (The Snap To Grid option works only for mouse operations.) Similarly, by pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. (If you press Shift+Ctrl along with an arrow key, the component will be moved only at grid intervals.) Unfortunately, during these fine-tuning operations, the component hints with the position and size are not displayed.

To align multiple components or make them the same size, you can select several components and set the Top, Left, Width, or Height property for all of them at the same time. To select several components, you can click them with the mouse while holding down the Shift key, or, if all the components fall into a rectangular area, you can drag the mouse to "draw" a rectangle surrounding them. When you've selected multiple components, you can also set their relative position using the Alignment dialog box (with the Align command of the form's shortcut menu) or the Alignment Palette (accessible through the View ➢ Alignment Palette menu command).

When you've finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component in a form. This is particularly helpful, as Undo operations on forms are limited (only an Undelete one), but the setting is not persistent.

Among its other features, the Form Designer offers several Tooltip hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. Delphi 6 offers extended hints, with details on the control position, size, tab order, and more. This is an addition to the Show Component Captions environment setting, which I keep active.
- As you resize a control, the hint shows the current size (the Width and Height properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the Left and Top properties).

Finally, you can save DFM (Delphi Form Module) files in plain text instead of the traditional binary resource format. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the

Designer page of the Environment Options dialog box. In the same page, you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box).

Having DFM files stored as text was a welcome addition in Delphi 5; it lets you operate more effectively with version-control systems. Programmers won't get a real advantage from this feature, as you could already open the binary DFM files in the Delphi editor with a specific command of the shortcut menu of the designer. Version-control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file.

In any case, note that if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFMs are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded faster).

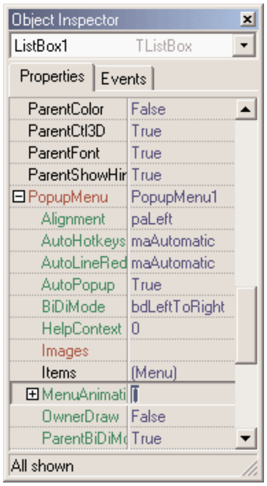
NOTE Text DFM files are more portable between versions of Delphi than their binary version. While an older version of Delphi might not accept a new property of a control in a DFM created by a newer version of Delphi, the older Delphis will still be able to read the rest of the text DFM file. If the newer version of Delphi adds a new data type, though, older Delphis will be unable to read the newer Delphi's binary DFMs at all. Even if this doesn't sound likely, remember that 64-bit operating systems are just around the corner. When in doubt, save in text DFM format. Also note that all versions of Delphi support text DFMs, using the command-line tool Convert in the `bin` directory.

The Object Inspector in Delphi 6

Delphi 5 provided new features to the Object Inspector, and Delphi 6 includes even more additions to it. As this is a tool programmers use all the time, along with the editor and the Form Designer, its improvements are really significant.

The most important change in Delphi 6 is the ability of the Object Inspector to expand component references in-place. Properties referring to other components are now displayed in a different color and can be expanded by selecting the + symbol on the left, as it happens with internal subcomponents. You can then modify the properties of that other component without having to select it. See Figure 1.6 for an example.

NOTE This interface-expansion feature also supports subcomponents, as demonstrated by the new LabeledEdit control.



Object Inspector

ListBox1 TListBox

Properties | Events

ParentColor	False
ParentCtl3D	True
ParentFont	True
ParentShowHir	True
⊟ PopupMenu	PopupMenu1
Alignment	paLeft
AutoHotkeys	maAutomatic
AutoLineRed	maAutomatic
AutoPopup	True
BiDiMode	bdLeftToRight
HelpContext	0
Images	
Items	(Menu)
⊞ MenuAnimati	[]
OwnerDraw	False
ParentBiDiMc	True

All shown

FIGURE 1.6: A connected component (a pop-up menu) expanded in the Object Inspector while working on another component (a list box)

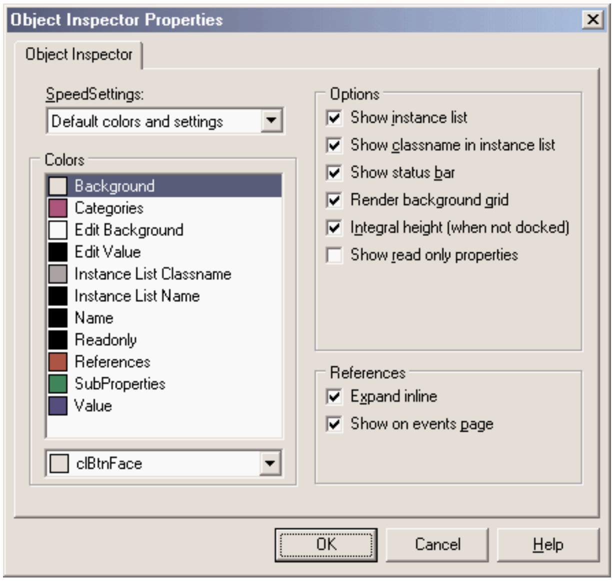
TIP A related feature of the Object Inspector is the ability to select the component referenced by a property. To do this, double-click the property value with the left mouse button while keeping the Ctrl key pressed. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the MainMenu property of the form and Ctrl+double-clicking the value of this property. This selects the main menu indicated as the value of the property in the Object Inspector.

Here are some other relevant changes of the Object Inspector:

- The list at the top of the Object Inspector shows the type of the object and can be removed to save some space (and considering the presence of the Object TreeView).
- The properties that reference an object are now a different color and may be expanded without changing the selection.
- You can optionally also view read-only properties in the Object Inspector. Of course, they are grayed out.
- The Object Inspector has a new Properties dialog box (see Figure 1.7), which allows you to customize the colors of the various types of properties and the overall behavior of this window.

FIGURE 1.7:

The new Object Inspector Properties dialog box



Object Inspector Properties

Object Inspector

SpeedSettings: Default colors and settings

Colors
- Background
- Categories
- Edit Background
- Edit Value
- Instance List Classname
- Instance List Name
- Name
- Readonly
- References
- SubProperties
- Value

clBtnFace

Options
- ☑ Show instance list
- ☑ Show classname in instance list
- ☑ Show status bar
- ☑ Render background grid
- ☑ Integral height (when not docked)
- □ Show read only properties

References
- ☑ Expand inline
- ☑ Show on events page

- Since Delphi 5, the drop-down list for a property can include graphical elements. This is used for properties such as `Color` and `Cursor`, and is particularly useful for the `ImageIndex` property of components connected with an ImageList.

NOTE

Interface properties can now be configured at design time using the Object Inspector. This makes use of the Interfaced Component Reference model introduced in Kylix/Delphi 6, where components may implement and hold references to interfaces as long as the interfaces are implemented by components. Interfaced Component References work like plain old component references, except that interface properties can be bound to any component that implements the necessary interface. Unlike component properties, interface properties are not limited to a specific component type (a class or its derived classes). When you click the drop-down list in the Object Inspector editor for an interface property, all components on the current form (and linked forms) that implement the interface are shown.

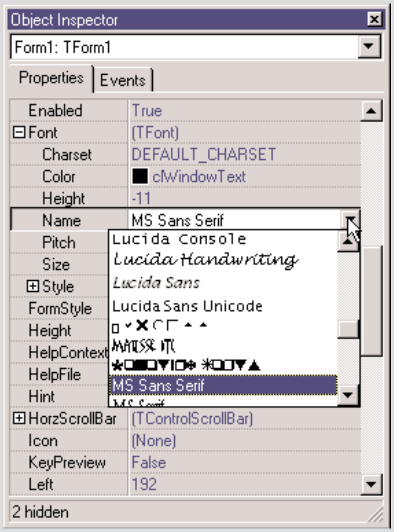
Drop-Down Fonts in the Object Inspector

The Delphi Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting, corresponding to the `Name` subproperty of the `Font` property. This capability is actually built into Delphi, but it has been disabled because most computers have a large number of fonts installed and rendering

Continued on next page

them can really slow down the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplayFontNames` global variable of the new VCLEditors unit. I've done this in the OiFontPk package, which you can find among the program examples for this chapter on the companion CD-ROM.

Once this package is installed, you can move to the `Font` property of any component and use the graphical Name drop-down menu, as displayed here:



Object Inspector

Form1: TForm1

Properties | Events

Enabled	True
⊟Font	(TFont)
Charset	DEFAULT_CHARSET
Color	■ clWindowText
Height	-11
Name	MS Sans Serif
Pitch	
Size	
⊞Style	
FormStyle	
Height	
HelpContext	
HelpFile	
Hint	
⊞HorzScrollBar	(TControlScrollBar)
Icon	(None)
KeyPreview	False
Left	192

Lucida Console
Lucida Handwriting
Lucida Sans
Lucida Sans Unicode
MS Sans Serif
MS Serif

2 hidden

There is a second, more complex customization of the Object Inspector that I like and use frequently: a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations. You can find the package to install custom fonts in the Object Inspector on my Web site, `www.marcocantu.com`.

Property Categories

Delphi 5 also introduced the idea of property categories, activated by the Arrange option of the local menu of the Object Inspector. If you set it, properties won't be listed alphabetically but arranged by group, with each property possibly appearing in multiple groups.

Categories have the benefit of reducing the complexity of the Object Inspector. You can use the View submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories).

Secrets of the Component Palette

The Component Palette is very simple to use, but there are a few things you might not know. There are four simple ways to place a component on a form:

- After selecting a control in the palette, click within the form to set the position for the control, and press-and-drag the mouse to size it.
- After selecting any component, simply click within the form to place it with the default height and width.
- Double-click the icon in the palette to add a component of that type in the center of the form.
- Shift-click the component icon to place several components of the same kind in the form. To stop this operation, simply click the standard selector (the arrow icon) on the left side of the Component Palette.

You can select the Properties command on the shortcut menu of the palette to completely rearrange the components in the various pages, possibly adding new elements or just moving them from page to page. In the resulting Properties page, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page.

TIP When you have too many pages in the Component Palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: Rename the pages with shorter names, so that all the pages will fit on the screen. Obvious—once you've thought about it.

The real undocumented feature of the Component Palette is the "hot-track" activation. By setting special keys of the Registry, you can simply select a page of the palette by moving over the tab, without any mouse click. The same feature can be applied to the component scrollers on both sides of the palette, which show up when a page has too many components. To activate this hidden feature, you must add an `Extras` key under `HKEY_CURRENT_USER\Software\Borland\Delphi\6.0`. Under this key enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string '1'.

Defining Event Handlers

There are several techniques you can use to define a handler for an event of a component:

- Select the component, move to the Events page, and either double-click in the white area on the right side of the event or type a name in that area and press the Enter key.
- For many controls, you can double-click them to perform the default action, which is to add a handler for the `OnClick`, `OnChange`, or `OnCreate` events.

When you want to remove an event handler you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event handler that is still empty, consider adding a comment to it (even just the `//` characters), so that it will not be removed.

Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate the component in the form. During this operation, Delphi duplicates all the properties, keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

It is also possible to copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source-code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'My Button'
  TabOrder = 0
  Font.Name = 'Arial'
end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

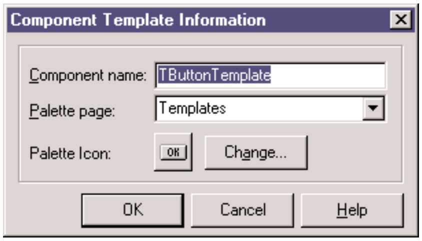
To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something completely wrong and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form, by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component ➢ Create Component Template menu command. This opens the Component Template Information dialog box, where you enter the name of the template, the page of the Component Palette where it should appear, and an icon.



Component Template Information

Component name: TButtonTemplate

Palette page: Templates

Palette Icon:

By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

All the information about component templates is stored in a single file, DELPHI32.DCT, but there is apparently no way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

TIP A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry `CCLibDir` under the key `\Software\Borland\Delphi\6.0\Component Templates`.

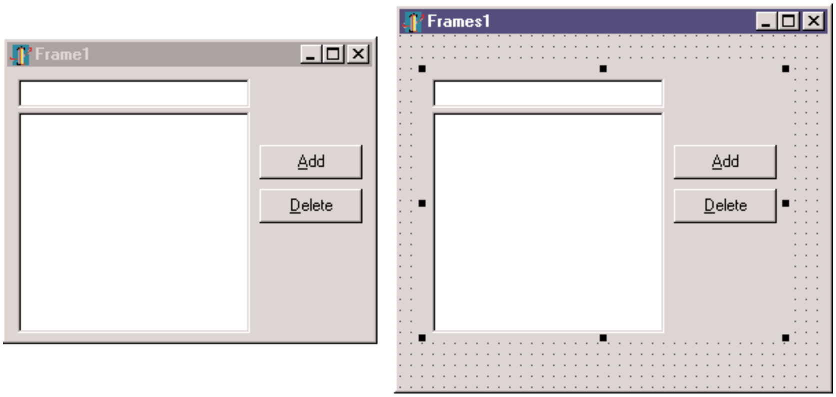
Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the *frames* technology in Delphi does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready, you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each of the instances of the frame.

You can see a simple example, called Frames1, in Figure 1.8 (its code is available on the companion CD). A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames. Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than component templates. Chapter 4 provides an in-depth look at VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful new technique.

FIGURE 1.8:

The Frames1 example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.



Managing Projects

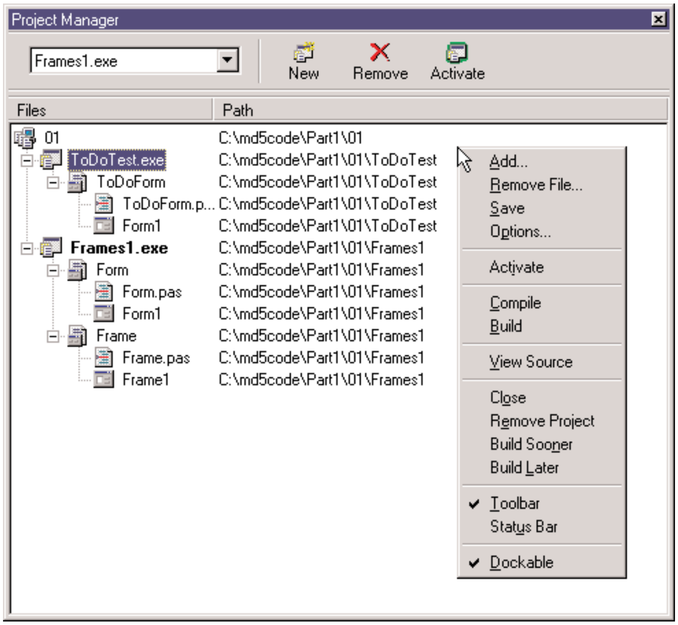
Delphi's multitarget Project Manager (View ➢ Project Manager) works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files.

TIP In Delphi 6, all open packages will show up as projects in the Project Manager view, even if they haven't been added to the project group.

In Figure 1.9, you can see the Project Manager with the project group for the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all of the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on it. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, or to open a unit.

Of all the projects in the group, only one is active, and this is the project you operate upon when you select a command such as Project ➢ Compile. The Project pull-down of the main menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.) When you have multiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.

FIGURE 1.9:
Delphi's multitarget Project Manager



Files	Path
01	C:\md5code\Part1\01
ToDoTest.exe	C:\md5code\Part1\01\ToDoTest
ToDoForm	C:\md5code\Part1\01\ToDoTest
ToDoForm.p...	C:\md5code\Part1\01\ToDoTest
Form1	C:\md5code\Part1\01\ToDoTest
Frames1.exe	C:\md5code\Part1\01\Frames1
Form	C:\md5code\Part1\01\Frames1
Form.pas	C:\md5code\Part1\01\Frames1
Form1	C:\md5code\Part1\01\Frames1
Frame	C:\md5code\Part1\01\Frames1
Frame.pas	C:\md5code\Part1\01\Frames1
Frame1	C:\md5code\Part1\01\Frames1

Among its advanced features, you can drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project.

The Project Manager automatically selects as the current project the one you are working with—for example, opening a file. You can easily see which project is selected and change it by using the combo box on the top of the form.

TIP

Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu, and choose Resource File (*.rc) as the file type. This resource file will be automatically bound to the project, even without a corresponding `$R` directive.

Delphi saves the project groups with the new .BPG extension, which stands for Borland Project Group. This feature comes from C++Builder and from past Borland C++ compilers, a history that is clearly visible as you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment. Here is a simple example:

```
#------------------------------
VERSION = BWS.01
#------------------------------
!ifndef ROOT
ROOT = $(MAKEDIR)\..
```

```
!endif
#------------------------------
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)\bin\dcc32.exe $**
BRCC = $(ROOT)\bin\brcc32.exe $**
#------------------------------
PROJECTS = Project1.exe
#------------------------------
default: $(PROJECTS)
#------------------------------
Project1.exe: Project1.dpr
  $(DCC)
```

Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. What you can do instead is invoke the Project Options dialog from the Project Manager for each project. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{$X+}` or `{$X-}` commands, where you'd replace *X* with the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All project options are saved automatically with the project, but in a separate file with a .DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a CFG file, for command-line compilation. The two files have similar content but a different format: The *dcc* command-line compiler, in fact, cannot use .DOF files, but needs the .CFG format.

Another alternative for saving compiler options is to press Ctrl+O+O (press the O key twice while keeping Ctrl pressed). This inserts, at the top of the current unit, compiler directives that correspond to the current project options, as in the following listing:

```
{$A+,B-,C+,D+,E-,F-,G+,H+,I+,J+,K-,L+,M-,N+,O+,P+,Q-,R-,S-,T-,U-,V+,
W-,X+,Y+,Z1}

{$MINSTACKSIZE $00004000}
```

```
{$MAXSTACKSIZE $00100000}

{$IMAGEBASE $00400000}

{$APPTYPE GUI}
```

Compiling and Building Projects

There are several ways to compile a project. If you run it (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed.

If you select Compile ➢ Build All instead, every file is compiled, even if it has not changed. You should only need this second command infrequently, since Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options, in which case you have to use the Build All command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi compiled unit (DCU). (This step is performed only if the DCU file is not already up-to-date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option (in the Preferences page of the Environment Options dialog box).

WARNING Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for the cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows some strange error message, the first thing you should try is the Build All command to resynchronize the make feature with the current files on disk.

The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't run unless the proper dynamic link libraries (such as `vcl50.bpl`, which is quite large) are available on the computer where you want to run the program.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course, if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in Chapter 12.

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semicompiled" tools.

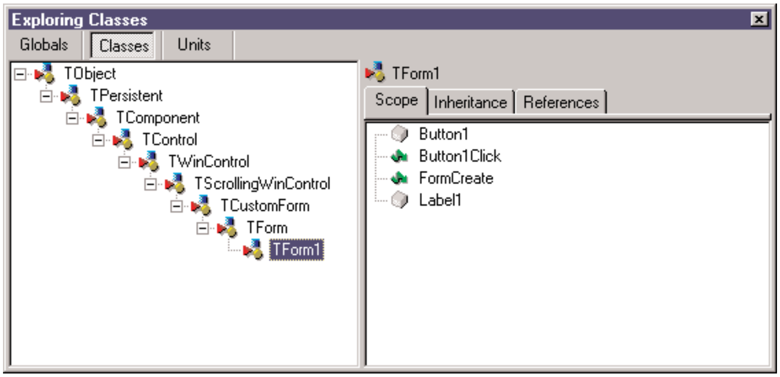
Exploring a Project

Past versions of Delphi included an Object Browser, which you could use when a project was compiled to see a hierarchical structure of its classes and to look for its symbols and the source-code lines where they are referenced. Delphi now includes a similar but enhanced tool, called Project Explorer. Like the Code Explorer, it is updated automatically as you type, without recompiling the project.

The Project Explorer allows you to list Classes, Units, and Globals, and lets you choose whether to look only for symbols defined within your project or for those from both your project and VCL. You can see an example with only project symbols in Figure 1.10.

FIGURE 1.10:

The Project Explorer



You can change the settings of this Explorer and those of the Code Explorer in the Explorer page of the Environment Options or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the Explorer categories you see in this window are specific to the Project Explorer; others relate to both tools.

Additional and External Delphi Tools

Besides the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (`PCE.exe`), and the Image Editor (`ImagEdit.exe`), are available from Tools menu of the IDE. In addition, the Enterprise edition has a link to the SQL Monitor (`SqlMon.exe`).

Other tools that are not directly accessible from the IDE include many command-line tools you can find in the `bin` directory of Delphi. For example, there is a command-line Delphi compiler (`DCC.exe`), a Borland resource compiler (`BRC32.exe` and `BRCC32.exe`), and an executable viewer (`TDump.exe`).

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools, mostly available in the `\Delphi6\bin` folder and in the Tools menu:

Web App Debugger (`WebAppDbg.exe`) is the debugging Web server introduced in Delphi 6. It is used to keep track of the requests send to your applications and debug them. I'll discuss this tool in Chapter 21.

XML Mapper (`XmlMapper.exe`), again new in Delphi 6, is a tool for creating XML transformations to be applied to the format produced by the ClientDataSet component. More on this topic in Chapter 22.

External Translation Manager (`etm60.exe`) is the stand-alone version of the Integrated Translation Manager. This external tool can be given to external translators and is available for the first time in Delphi 6.

Borland Registry Cleanup Utility (`D6RegClean.exe`) helps you remove all of the Registry entries added by Delphi 6 to a computer.

TeamSource is an advanced version-control system provided with Delphi, starting with version 5. The tool is very similar to its past incarnation and is installed separately from Delphi.

WinSight (Ws.exe) is a Windows "message spy" program available in the bin directory.

Database Explorer can be activated from the Delphi IDE or as a stand-alone tool, using the DBExplor.exe program of the bin directory.

OpenHelp (oh.exe) is the tool you can use to manage the structure of Delphi's own Help files, integrating third-party files into the help system.

Convert (Convert.exe) is a command-line tool you can use to convert DFM files into the equivalent textual description and vice versa.

Turbo Grep (Grep.exe) is a command-line search utility, much faster than the embedded Find In Files mechanism but not so easy to use.

Turbo Register Server (TRegSvr.exe) is a tool you can use to register ActiveX libraries and COM servers. The source code of this tool is available under \Demos\ActiveX\TRegSvr.

Resource Explorer is a powerful resource viewer (but not a full-blown resource editor) you can find under \Demos\ResXplor.

Resource Workshop The Delphi 5 CD also includes a separate installation for Resource Workshop. This is an old 16-bit resource editor that can also manage Win32 resource files. It was formerly included in Borland C++ and Pascal compilers for Windows and was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long filenames, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files.

The Files Produced by the System

Delphi produces various files for each project, and you should know what they are and how they are named. Basically, two elements have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you'll find in the directory where a Delphi project resides. The table also shows when or under what circumstances these files are created and their importance for future compilations.

TABLE 1.1: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to end-users.
.CAB	The Microsoft Cabinet compressed-file format used for Web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.
.DCP	Delphi Component Package: a file with symbol information for the code that was compiled into the package. It doesn't include compiled code, which is stored in DCU files.	Compilation	Required when you use packages. You'll distribute it only to other developers along with DPL files.
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DDP	The new Delphi Diagram Portfolio, used by the Diagram view of the editor (was .DTI in Delphi 5)	Development	No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.

TABLE 1.1 continued: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
.~DF	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN	Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).	Development (ITE)	Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.
.DLL	Dynamic Link Library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK	Delphi Package: the project source code file of a package.	Development	Yes.
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
.~DP	Backup of the Delphi Project file (.DPR).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.

TABLE 1.1 continued: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.DSM	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.EXE	Executable file: the Windows application you've produced.	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or .HTML, for Hypertext Markup Language: the file format used for Internet Web pages.	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
OCX	OLE Control Extension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
.PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.
.~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.

TABLE 1.1 continued: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.RPS	Translation Repository (part of the Integrated Translation Environment).	Development (ITE)	No. Required to manage the translations.
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
TODO	To-do list file, holding the items related to the entire project.	Development	No. This file hosts notes for the programmers.
.UDL	Microsoft Data Link.	Development	Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).

Besides the files generated during the development of a project in Delphi, there are many others generated and used by the IDE itself. In Table 1.2, I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

TABLE 1.2: Selected Delphi IDE Customization File Extensions

Extension	File Type
.DCI	Delphi code templates
.DRO	Delphi's Object Repository (The repository should be modified with the Tools ➢ Repository command.)
.DMT	Delphi menu templates
.DBI	Database Explorer information
.DEM	Delphi edit mask (files with country-specific formats for edit masks)
.DCT	Delphi component templates
.DST	Desktop settings file (one for each desktop setting you've defined)

Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time covering their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depends on syntax highlighting, but it isn't saved with the file. It is worth noting that there is one single file for the whole code of the form, not just small code fragments.

TIP In the listings in this book, I've matched the bold syntax highlighting of the editor for keywords and the italic for strings and comments.

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a .DFM extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is a binary and, in Delphi, can be saved either as a plain-text file or in the traditional Windows Resource format. You can set the default format you want to use for new projects in the Designerpage of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command of a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can actually edit the textual description of a form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, compilation will stop with an error message and you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have good knowledge of Delphi programming.

TIP In the book, I often show you excerpts of DFM files. In most of these excerpts, I only show the most relevant components or properties; generally, I have removed the positional properties, the binary values, and other lines providing little useful information.

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the Project -> View Source menu command.

Some of the other, less relevant files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF):

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

NOTE A lot of information related to the status of the Delphi environment is saved in the Windows Registry, as well as in DSK and other files. I've already indicated a few special undocumented entries of the Registry you can use to activate specific features. You should explore the `HKEY_CURRENT_USER\Software\Borland\Delphi\6.0` section of the Registry to examine all the settings of the Delphi IDE (including all those you can modify with the Project Options and the Environment Options dialog boxes, as well as many others).

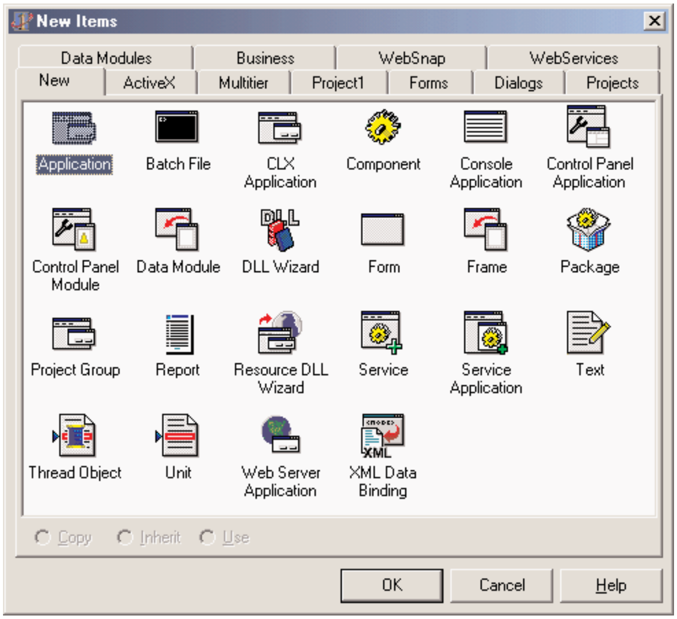
The Object Repository

Delphi has menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File ➢ New menu and in other pull-down menus. What happens if you simply select File ➢ New ➢ Other? Delphi opens the Object Repository, which is used to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New dialog box (shown in Figure 1.11) has several pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.

FIGURE 1.11:

The first page of the New dialog box, generally known as the "Object Repository"



TIP The Object Repository has a shortcut menu that allows you to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool, information that is particularly important when looking at wizards, projects, or forms that you've added to the Repository.

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and "current project" pages). Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, ready to use later on. Simply use the Project ➢ Add To Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form that you want to add and select the Add To Repository command of its shortcut menu. Then indicate the title, description, author, page, and icon in its dialog box.

You might want to keep in mind that as you copy a project or form template to the repository and then copy it back to another directory, you are simply doing a copy and paste operation. This isn't much different than copying the files manually.

The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or Wizards, this is not what you want, however. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple:

1. Create a new project as usual.
2. Remove its only form from the project.
3. Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you have to remember to use the File ➢ Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools ➢ Repository command. This opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing ones. You can even add new pages, rename or delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File ➢ New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File ➢ New Application) when no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File ➢ New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the `OBJREPOS` subdirectory of the Delphi main directory. At the same time, if you use a form or any other object directly without copying it, then you end up having some files of your project in this directory. It is important to realize how the Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files, without copying data back and forth to the Repository.

Installing New DLL Wizards

Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they would be installed the same way you install a component or a package. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key `\Software\Borland\Delphi\6.0\Experts`. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.

What's Next?

This chapter has presented an overview of the new and more advanced features of the Delphi 6 programming environment, including tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to start *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

Now we are ready to spend the next two chapters looking into the Object Pascal language and then proceed by studying the RTL and the class library included in Delphi 6.

CHAPTER 2

The Object Pascal Language: Classes and Objects

- The Pascal language
- New conditional compilation and hint directives
- Classes and objects
- The *Self* keyword
- Class methods and overloading
- Encapsulation: *private* and *public*
- Using properties
- Constructors
- Objects and memory

Most modern programming languages support *object-oriented programming* (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding).

You can write Delphi applications even without knowing the details of Object Pascal. As you create a new form, add new components, and handle events, Delphi prepares most of the related code for you automatically. But knowing the details of the language and its implementation will help you to understand precisely what Delphi is doing and to master the language completely.

A single chapter doesn't allow space for a full introduction to the principles of object-oriented programming and the Object Pascal language. Instead, I will outline the key OOP features of the language and show how they relate to everyday Delphi programming. Even if you don't have a precise knowledge of OOP, the chapter will introduce each of the key concepts so that you won't need to refer to other sources.

The Pascal Language

The Object Pascal language used by Delphi is an OOP extension of the classic Pascal language, which Borland pushed forward for many years with its Turbo Pascal compilers. The syntax of the Pascal language is known to be quite verbose and more readable than, for example, the C language. Its OOP extension follows the same approach, delivering the same power of the recent breed of OOP languages, from Java to C#.

In this chapter, I'll discuss only the object-oriented extensions of the Pascal language available in Delphi. However, I'll highlight recent additions Borland has done to the core language. These features have been introduced in Delphi 6 and are, at least partially, related to the Linux version of Delphi.

New Pascal features include the `$IF` and `$ELSEIF` directives for conditional compilation, the `$WARN` and `$MESSAGE` directives, and the `platform`, `library`, and `deprecated` hint directives. These topics are discussed in the following sections. Changes to the assembler (with new directives, support for MMX and Pentium Pro instructions, and many more features) are really beyond the scope of this book.

Other relatively minor changes in the language include a change in the default value for the `$WRITEABLECONST` compiler switch, which is now disabled. This option allows programs to modify the value of typed constants and should generally be left disabled, using variables instead of constants for modifyable values. Another change is the support for the `Int64` data type in variants. Finally, you can assign specific values to the elements of an enumeration (as in the C/C++ language), instead of using the default sequence of values.

The New *$IF* Compiler Directive

Delphi has always had a $IFDEF directive you could use to test whether a specific symbol was defined. (Delphi also has a $IFNDEF directive, with the opposite test.) This is used to obtain conditional compilation, as in

```
{$IFDEF DEBUG}
  // executes only if the DEBUG directive is set
  ShowMessage ('Executing critical code');
{$ENDIF}
```

By setting or not setting the DEBUG directive and recompiling, the extra line of code will be included or skipped by the compiler.

This code directive is powerful, but checking for multiple versions of Delphi and operating systems can force you to use multiple-nested $IFDEF directives, making the code totally unreadable. For this reason, Borland has introduced a new and more powerful directive for conditional compilation, $IF. Inside the directive you can use the Defined function to check whether a conditional symbol is defined, or use the Declared function to see whether a language constant is defined and use these constants within a constant Boolean expression. Here is some code that shows how to use a constant within the $IF directive (you can find this and other code excerpts of this and the next section in the IfDirective example on the companion CD):

```
const
  DebugControl = 2;

{$IF Defined(DEBUG) and (DebugControl > 3)}
  ShowMessage ('Executing critical code');
{$IFEND}
```

Notice that the statement is closed by a $IFEND and that you can also have an optional $ELSE branch. You can also concatenate conditions with the $ELSEIF directive, followed by another condition and evaluated only as an alternative to the $IF directive it refers to:

```
{$IF one}
  ...
{$ELSEIF two}
  ...
{$ELSE}
  ...
{$IFEND}
```

Within the expressions of the $IF directive, you can use only untyped constants, which are really and invariably treated as constants by the compiler. You can follow the general rules of Pascal constant expressions. You can use all the language operators, the and, or, xor, and not Boolean operators, and mathematical ones including div, mod, +, -, *, /, > and <, to mention just a few common ones. You can also use predefined functions such as SizeOf, High, Low, Prev, Succ, and others listed in the Delphi Help page “Constant expressions.” The expression can use constant symbols of any type, including floats and strings, so long as the expression itself ultimately evaluates to a True or False value.

WARNING In these constant expressions, it is not possible to use type constants, which can be optionally modified in the code depending on the status of the writeable-typed constants directive ($J or $WRITEABLECONST). In any case, using constants you can modify is quite a bad idea in the first place.

Delphi provides a few predefined conditional symbols, including compiler version, the operating system, the GUI environment, and so on. I’ve listed the most important ones in Table 2.1. You can also use the RTLVersion constant defined in the System unit to test which version of Delphi (and its run-time library) you are compiling on. The predefined symbol ConditionalExpressions can be used to shield the new directives from older versions of Delphi:

```
{$IFDEF ConditionalExpressions}
   {$IF System.RTLVersion > 14.0}
      // do something
   {$IFEND}
{$ENDIF}
```

TABLE 2.1: Commonly Used Predefined Conditional Symbols

Symbol	Description
VER140	Compiling with Delphi 6, which is the 14.0 version of the Borland Pascal compiler; Delphi 5 used VER130, with lower numbers for past versions.
MSWINDOWS	Compiling on the Windows platform (new in Delphi 6).
LINUX	Compiling on the Linux platform. On Kylix, there are also the LINUX32, POSIX, and ELF predefined symbols.
WIN32	Compiling only on the 32-bit Windows platform. This symbol was introduced in Delphi 2 to distinguish from 16-bit Windows compilations (Delphi 1 defined the WINDOWS symbol). You should use WIN32 only to mark code specifically for Win32, not Win16 or future Win64 platforms (for which the WIN64 symbol has been reserved). Use MSWINDOWS, instead, to distinguish between Windows and other operating systems.
CONSOLE	Compiling a console application, and not a GUI one. This symbol is meaningful only under Windows, as all Linux applications are console applications.
BCB	Defined when the C++Builder IDE invokes the Pascal compiler.
ConditionalExpressions	Indicates that the $IF directive is available. It is defined in Kylix and Delphi 6, but not in earlier versions.

TIP I recommend using conditional compilation sparingly and only when it is really required. It is generally better, whenever possible, to write code that can adapt to different situations—for example, adding different versions of the same class (or different inherited classes) to the same program. Excessive use of conditional compilation makes a program hard to read and to debug.

WARNING Remember to issue a Build All command when you change a conditional symbol or a constant, which can affect a conditional compilation; otherwise the affected units won't be recompiled unless their source code changes.

New Hint Directives

Supporting multiple operating systems within the same source code base implies a number of compatibility issues. Besides a modified run-time library and a wholly new component library (discussed in Chapter 4, "The Run-Time Library," and Chapter 5, "Core Library Classes"), Delphi 6 includes special directives Borland uses to mark special portions of the code. As they introduced the idea of custom warnings and messages (described in the previous section), they've added a few special predefined ones.

The *platform* Directive

The first directive of this group is the `platform` directive, used to mark nonportable code. This directive can be used to mark procedures, variables, types, and almost any defined symbol. Borland uses `platform` in its libraries, so that when you use a platform-specific capability (for example, calling the `IncludeTrailingBackslash` function of the `SysUtils` unit), you'll receive a warning message, such as:

```
Symbol 'IncludeTrailingBackslash' is specific to a platform.
```

This warning is a hint for developers who plan to port their code between the Linux and Windows platforms, even in the future. In many cases, you'll be able to find an alternative approach that is fully platform independent. Check the help file (or eventually the library source code) for hints in this direction. In the case of the `IncludeTrailingBackslash` function, there is now a new version, called `IncludeTrailingDelimiter`, that is also portable to a Unix-based file system.

Of course you can use the `platform` directive to mark your code, for example, if you write a component or library that has platform-specific features. Here are a few examples:

```
var
  windowsversion: Integer = 2000 platform;
```

```
procedure Test; platform;
begin
  Beep;
end;

type
  TWinClass = class
    x: Integer;
  end platform;
```

The code fragments of this section are available, for your experiments, in the IfDirective example on the companion CD.

NOTE The position of semicolons for hint directives can be quite confusing at first. The rule is that a hint directive must appear before the semicolon following the symbol it modifies. But a procedure, function, or unit header declaration can be followed only by reserved words, so its hint directive can appear following the semicolon. A type, variable, or constant declaration can be followed by another identifier, so the hint directive must come before the semicolon closing its declaration. Part of the rationale behind this is that the hint directives are not reserved words, so they can be used as the name of an identifier.

The *deprecated* Directive

The `deprecated` directive works in a similar way to the `platform` directive; the only real differences are that it is used in a different context and produces a different compiler warning. The role of `deprecated` is to mark identifiers that are still part of the system for compatibility reasons, but either are going to be removed in the future or expose you to risks of incompatibility. This symbol is used sparingly in the Delphi library.

The *library* Directive

The `library` directive works in a similar way to `deprecated` and `platform`; its role is to mark out code or components that are specific to a library (either VCL or CLX) and are not portable among them. However, apparently this symbol is never used within the Delphi library.

The *$WARN* Directive

The `$WARNINGS` directive (and the corresponding compiler option) allows you to turn off all the warning messages. Most programmers like to keep the messages on and tend to work with programs that compile with no hints and warnings. With the advent of the three hint directives discussed in the last section, however, there are programs specifically aimed for a platform, which cannot compile without compatibility warnings.

To overcome this situation, Delphi 6 introduces the $WARN directive, specifically aimed at disabling hint directives. As an example, you'll disable platform hints by writing this code:

```
{$WARN SYMBOL_PLATFORM OFF}
```

The $WARN directive has five different parameters, related to the three hint directives, and can use the ON and OFF values for each:

- SYMBOL_PLATFORM and UNIT_PLATFORM can be used to disable the platform directive in the current unit or in the unit where the directive is specified. The warning, in fact, is issued while compiling the code that uses the symbol, not while compiling the code with the definition.
- SYMBOL_LIBRARY and UNIT_LIBRARY work on the library directive in the same manner as the platform-related parameters above.
- SYMBOL_DEPRECATED can be used to disable the deprecated directive.

The *$MESSAGE* Directive

The compiler has now the ability to generate warnings in many different situations, so that the developer of a library or a portion of a program can let other programmers know of a given problem or risk in using a given feature, when the program can still legally compile. An extension to this idea is to let programmers insert custom warning messages in the code, with this syntax:

```
{$MESSAGE 'Old version of the unit: consider using the updated version'}
```

Compiling this code will issue a hint message with the text provided. This feature can be used to indicate possible problems, suggest alternative approaches, mark unfinished code, and more. This is probably more reliable than using a TODO item (discussed in the preceding chapter), because a programmer might not open the To-Do List window but the compiler will remind him of the pending problem. However, it is the compiler that issues the message, so you'll see it even if the given portion of the code is not really used by the program because the linker will remove it from the executable file.

These type of free messages, like the hint directives, become very useful to let the developer of a component communicate with the programmers using it, warning of potential pitfalls.

Introducing Classes and Objects

The cornerstone of the OOP extensions available in Object Pascal is represented by the class keyword, which is used inside type declarations. Classes define the blueprint of the objects you create in Delphi. As the terms *class* and *object* are commonly used and often misused, let's be sure we agree on their definitions.

A *class* is a user-defined data type, which has a state (its representation) and some operations (its behavior). A class has some internal data and some methods, in the form of procedures or functions, and usually describes the generic characteristics and behavior of some similar objects.

An *object* is an instance of a class, or a variable of the data type defined by the class. Objects are *actual* entities. When the program runs, objects take up some memory for their internal representation. The relationship between object and class is the same as the one between variable and type.

To declare a new class data type in Object Pascal, with some local data fields and some methods, use the following syntax:

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

NOTE The convention in Delphi is to use the letter *T* as a prefix for the name of every class you write and every other type (*T* stands for *Type*). This is just a convention—to the compiler, *T* is just a letter like any other—but it is so common that following it will make your code easier to understand.

The following is a complete class definition, with two methods declared and not yet fully defined. The definition of these two methods (the `LeapYear` function and the `SetValue` procedure) must be present in the same unit of the class declaration and are written with this syntax:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils.pas
  Result := IsLeapYear (Year);
end;
```

The method names are prefixed with the class name (using the dot-notation), because a unit can hold multiple classes, possibly with methods having the same names. You can actually avoid retyping the method names and parameter list by using the class completion feature of the editor. Simply type or modify the class definition and press Ctrl+Shift+C while the cursor is within the class definition itself; this will allow Delphi to generate a skeleton of the definition of the methods, including the `begin` and `end` statements.

Once the class has been defined, we can create an object and use it as follows:

```
var
  ADay: TDate;
begin
  // create an object
  ADay := TDate.Create;
  // use the object
  ADay.SetValue (1, 1, 2000);
  if ADay.LeapYear then
    ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
  // destroy the object
  ADay.Free;
end;
```

Notice that `ADay.LeapYear` is an expression similar to `ADay.Year`, although the first is a function call and the second a direct data access. You can optionally add parentheses after the call of a function with no parameters. You can find the code snippets above in the source code of the Date1 example; the only difference is that the program creates a date based on the year provided in an edit box.

Classes, Objects, and Visual Programming

When I teach classes about OOP in Delphi, I always tell my students that regardless of how much OOP you know and how much you use it, Delphi forces you in the OOP direction. Even if you simply create a new application with a form and place a button over it to execute some code when the button is pressed, you are building an object-oriented application. In fact, the form is an object of a new class (by default `TForm1`, which inherits from the base `TForm` class provided by Borland), and the button is an instance of the `TButton` class, provided by Borland, as you can see in the following code snippet:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  end;
```

Given these premises, it would be very hard to build a Delphi application without using classes and objects. Yes, I know it is technically possible, but I doubt it would make a lot of sense. Not using objects and classes with Delphi would probably be more difficult than using them, as you have to give up all of the design-time tools for visual programming.

In any case, the real challenge is using OOP properly, something I'll try to teach you in this chapter (and in the rest of the book), along with an introduction to the key elements of the Object Pascal language.

The *Self* Keyword

Methods are very similar to procedures and functions. The real difference is that methods have an implicit parameter, which is a reference to the current object. Within a method you can refer to this parameter—the current object—using the `Self` keyword. This extra hidden parameter is needed when you create several objects of the same class, so that each time you apply a method to one of the objects, the method will operate only on its own data and not affect sibling objects.

For example, in the `SetValue` method of the `TDate` class, listed earlier, we simply use `Month`, `Year`, and `Day` to refer to the fields of the current object, something you might express as

```
Self.Month := m;
Self.Day := d;
```

This is actually how the Delphi compiler translates the code, *not* how you are supposed to write it. The `Self` keyword is a fundamental language construct used by the compiler, but at times it is used by programmers to resolve name conflicts and to make tricky code more readable.

NOTE The C++ and Java languages have a similar feature based on the keyword `this`.

All you really need to know about `Self` is that the technical implementation of a call to a method differs from that of a call to a generic subroutine. Methods have an extra hidden parameter, `Self`. Because all this happens behind the scenes, you do not need to know how `Self` works at this time.

If you look at the definition of the `TMethod` data type in the System unit, you'll see that it is a record with a `Code` field and a `Data` field. The first is a pointer to the function's address in memory; the second the value of the `Self` parameter to use when calling that function address. We'll discuss method pointers in Chapter 5.

Overloaded Methods

Object Pascal supports overloaded functions and methods: you can have multiple methods with the same name, provided that the parameters are different. By checking the parameters, the compiler can determine which of the versions of the routine you want to call.

There are two basic rules:

- Each version of the method must be followed by the `overload` keyword.
- The differences must be in the number or type of the parameters or both. The return type cannot be used to distinguish between two methods.

Overloading can be applied to global functions and procedures and to methods of a class. As an example of overloading, I've added to the `TDate` class two different versions of the `SetValue` method:

```
type
  TDate = class
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
...//the rest of the class declaration

procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

procedure TDate.SetValue(NewDate: TDateTime);
begin
  fDate := NewDate;
end;
```

NOTE In Delphi 6, the compiler has been enhanced to improve the resolution of overloaded methods, allowing the compilation of calls that were considered ambiguous. In particular, the compiler handles the difference between `AnsiString` and `WideString` types. The overload resolution also has better support for variant-type parameters (which will provide matches in case there is no exact match for another overloaded version) and interfaces (which are given precedence to object types). Finally, the compiler allows the `nil` value to match an interface-type parameter. Some of these improvements were already introduced in the Kylix compiler.

Creating Components Dynamically

In Delphi, the `Self` keyword is often used when you need to refer to the current form explicitly in one of its methods. The typical example is the creation of a component at run time, where you must pass the owner of the component to its `Create` constructor and assign the same value to its `Parent` property. (The difference between `Owner` and `Parent` properties is discussed in the next chapter.) In both cases, you have to supply the current form as parameter or value, and the best way to do this is to use the `Self` keyword.

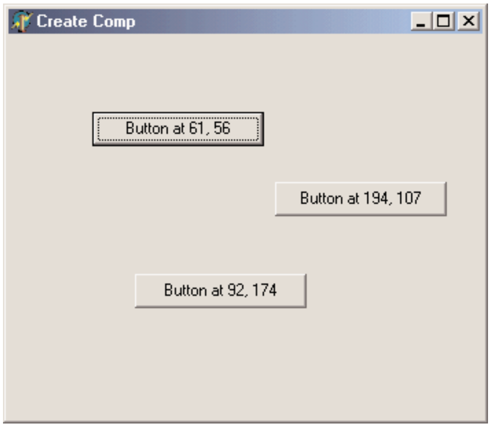
To demonstrate this kind of code, I've written the CreateC example (the name stands for *Create Component*) included on the companion CD. This program has a simple form with no components and a handler for its OnMouseDown event. I've used OnMouseDown because it receives as its parameter the position of the mouse click (unlike the OnClick event). I need this information to create a button component in that position. Here is the code of the method:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create (Self);
  Btn.Parent := Self;
  Btn.Left := X;
  Btn.Top := Y;
  Btn.Width := Btn.Width + 50;
  Btn.Caption := Format ('Button at %d, %d', [X, Y]);
end;
```

The effect of this code is to create buttons at mouse-click positions, with a caption indicating the exact location, as you can see in Figure 2.1. In the code above, notice in particular the use of the Self keyword, as the parameter of the Create method and as the value of the Parent property. I'll discuss these two elements (ownership and the Parent property) in Chapter 5.

FIGURE 2.1:

The output of the CreateC example, which creates Button components at run time



It is very common to write code like the above method using a with statement, as in the following listing:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

```
begin
  with TButton.Create (Self) do
  begin
    Parent := Self;
    Left := X;
    Top := Y;
    Width := Width + 50;
    Caption := Format ('Button in %d, %d', [X, Y]);
  end;
end;
```

TIP When writing a procedure like the code you've just seen, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't make any practical difference, but if there are multiple instances of a form, using `Form1` would be an error. In fact, if the `Form1` variable refers to the first form of that type being created, by clicking in another form of the same type, the new button will always be displayed in the first form. Its `Owner` and `Parent` will be `Form1` and not the form the user has clicked. In general, referring to a particular instance of a class when the current object is required is bad OOP practice.

Class Methods and Class Data

When you define a field in a class, you actually specify that the field should be added to each object of that class. Each instance has its own independent representation (referred to by the `Self` pointer). In some cases, however, it might be useful to have a field that is shared by all the objects of a class.

Other object-oriented programming languages have formal constructs to express this, while in Object Pascal we can simulate this feature using the encapsulation provided at the unit level. You can simply add a variable in the `implementation` portion of a unit, to obtain a class variable—a single memory location shared by all of the objects of a class.

If you need to access this value from outside the unit, you might use a method of the class. However, this forces you to apply this method to one of the instances of the class. An alternative solution is to declare a *class method*. A class method cannot access the data of any single object but can be applied to a class as a whole rather than to a particular instance.

To declare a class method in Object Pascal, you simply add the `class` keyword in front of it:

```
type
  MyClass = class
    class function ClassMeanValue: Integer;
```

The use of class methods is not very common in Object Pascal, because you can obtain the same effect by adding a procedure or function to a unit declaring a class. Object-oriented purists, however, will definitely prefer the use of a class method over a routine unrelated to a class. For example, an OOP purist would add a class method for getting the current date to a `TDate` class instead of using a global function (also because some OOP languages, including Java, don't have the notion of global functions).

We'll see several class methods in the next chapter, when we'll examine the structure of the `TObject` class.

TIP Contrary to other OOP languages, Delphi class methods can also be virtual, so they can be overridden and used to obtain polymorphism (a technique discussed later in this chapter).

Encapsulation

A class can have any amount of data and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated*, inside the class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, as the methods can check whether the date is valid and refuse to modify the new value if it is not. Encapsulation is important because it allows the class writer to modify the internal representation in a future version.

The concept of encapsulation is often indicated by the idea of a "black box," where you don't know about the internals: You only know how to interface with it or how to use it regardless of its internal structure. The "how to use" portion, called the *class interface*, allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Of course, you are supposed to use methods to access the data, which is shielded from unauthorized access. This is the object-oriented approach to a classical programming concept known as *information hiding*.

Delphi implements this class-based encapsulation but still supports the classic module-based encapsulation using the structure of units. Because the two are strictly related, let me recap the traditional approach first.

Encapsulation and Units

A unit in Object Pascal is a secondary source-code file, with the main source-code file being represented by the project source code. Every unit has two main sections, called `interface` and `implementation`, as well as two optional ones for `initialization` and `finalization` code. I want to focus here on the information hiding implemented by units.

In short, every identifier (type, routine, variable, and so on) that you declare in the interface portion of a unit becomes visible to any other unit of the program, provided there is a uses statement referring back to the unit that defines the identifier. All the routines and methods you declare in the interface portion of the unit must later be fully defined in the implemented portion of the same unit. In the interface section of a unit, however, you cannot write any actual statements to execute.

On the other hand, any identifier you declare in the implementation portion of the unit is local to the unit and is not visible outside it. A unit can have local data, local support functions, and even local types that the rest of the program is not allowed to access. This provides a direct way to hide the implementation details of an abstraction from its users, so you can later change your code without affecting other units of the program (and without even having to notify the changes to other programmers writing those units).

When you write classes in a unit, you'll generally define them in the interface portion of a unit, but some special keywords allow you to hide portions of this class interface.

Private, Protected, and Public

For class-based encapsulation, the Object Pascal language has three access specifiers: private, protected, and public. A fourth, published, controls RTTI and design time information and will be discussed in more detail in Chapter 5. Here are the three *classic* access specifiers:

- The private directive denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.
- The protected directive is used to indicate methods and fields with limited visibility. Only the current class and its subclasses can access protected elements. We'll discuss this keyword again in the "Protected Fields and Encapsulation" section.
- The public directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.

Generally, the fields of a class should be private; the methods are usually public. However, this is not always the case. Methods can be private or protected if they are needed only internally to perform some partial computation. Fields can be protected so that you can manipulate them in subclasses, but only if you are fairly sure that their type definition is not going to change. Access specifiers only restrict code outside your unit from accessing certain members of classes declared in the interface section of your unit. This means that if two classes are in the same unit, there is no protection for their private fields. Only by placing a class in the interface portion of a unit will you limit the visibility from classes and functions in other units to the public method and fields of the class.

As an example, consider this new version of the TDate class:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

In this version, the fields are now declared to be private, and there are some new methods. The first, GetText, is a function that returns a string with the date. You might think of adding other functions, such as GetDay, GetMonth, and GetYear, which simply return the corresponding private data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class you are implementing.

Another new method is the Increase procedure, which increases the date by one day. This is far from simple, because you need to consider the different lengths of the various months as well as leap and non–leap years. What I'll do to make it easier to write the code is change the internal implementation of the class to Delphi's TDateTime type for the internal implementation. The class definition will change to (the complete code will be in the next example, DateProp):

```
type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Notice that because the only change is in the private portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

NOTE The TDateTime type is actually a floating-point number. The integral portion of the number indicates the date since 12/30/1899, the same base date used by OLE Automation and Microsoft applications. (Use negative values to express previous years.) The decimal portion indicates the time as a fraction. For example, a value of 3.75 stands for the second of January 1900, at 6:00 A.M. (three-quarters of a day). To add or subtract dates, you can add or subtract the number of days, which is much simpler than adding days with a day/month/year representation.

Encapsulating with Properties

Properties are a very sound OOP mechanism, or a very well thought out application of the idea of encapsulation. Essentially, you have a name that completely hides its implementation details. This allows you to modify the class extensively without affecting the code using it. A good definition of properties is that of *virtual fields*. From the perspective of the user of the class that defines them, properties look exactly like fields, as you can generally read or write their value. For example, you can read the value of the Caption property of a button and assign it to the Text property of an edit box with the following code:

```
Edit1.Text := Button1.Caption;
```

This looks like we are reading and writing fields. However, properties can be directly mapped to data, as well as to access methods, for reading and writing the value. When properties are mapped to methods, the data they access can be part of the object or outside of it, and they can produce side effects, such as repainting a control after you change one of its values. Technically, a property is an identifier that is mapped to data or methods using a read and a write clause. For example, here is the definition of a Month property for a date class:

```
property Month: Integer read FMonth write SetMonth;
```

To access the value of the Month property, the program reads the value of the private field FMonth, while to change the property value it calls the method SetMonth (which must be defined inside the class, of course). Different combinations are possible (for example, we could also use a method to read the value or directly change a field in the write directive), but the use of a method to change the value of a property is very common. Here are two alternative definitions for the property, mapped to two access methods or mapped directly to data in both directions:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

TIP When you write code that accesses a property, it is important to realize that a method might be called. The issue is that some of these methods take some time to execute; they can also produce side effects, often including a (slow) repainting of the component on the screen. Although side effects of properties are seldom documented, you should be aware that they exist, particularly when you are trying to optimize your code.

Often, the actual data and access methods are private (or protected) while the property is public. This means you must use the property to have access to those methods or data, a technique that provides both an extended and a simplified version of encapsulation. It is an *extended* encapsulation because not only can you change the representation of the data and its access functions, but you can also add or remove access functions without changing the calling code at all. A user only needs to recompile the program using the property.

Class Completion for Properties

Properties provide a *simplified* encapsulation because when extra code is not required, you map the properties directly to fields, without writing tedious and useless access methods. And even when you want to write those methods, the IDE can use class completion (the Ctrl+Shift+C key combination) to generate the skeleton of the access methods of the properties for you. If you simply type in a class (say TMyClass),

```
property X: Integer;
```

and activate class completion, Delphi generates a SetX method for the property and adds the FX field to the class. The resulting code looks like this:

```
type
  TMyClass = class(TForm)
  private
    FX: Integer;
    procedure SetX(const Value: Integer);
  public
    property X: Integer read FX write SetX;
  end;

implementation

procedure TMyClass.SetX(const Value: Integer);
begin
  FX := Value;
end;
```

This really saves a lot of typing. You can even partially control how class completion generates Set and Get methods for the property. In fact, if you first type the property declaration including the read and write directives, as in

```
property X: Integer read GetX write SetX;
```

Class completion will generate the requested methods or add the field definition. If you want both the field and the methods, type in only the property name and its data type (as in the first example above), and let Delphi expand the declaration. At this point, fix the expanded declaration by replacing the FX field with a GetX method in the read portion, and invoke class completion a second time.

Properties for the *TDate* Class

As an example, I've added properties for accessing the year, the month, and the day to an object of the TDate class discussed earlier. These properties are not mapped to specific fields, but they all map to the single fDate field storing the entire date information. This is the new definition of the class:

```
type
  TDate = class
  private
    fDate: TDateTime;
    procedure SetDay(const Value: Integer);
    procedure SetMonth(const Value: Integer);
    procedure SetYear(const Value: Integer);
    function GetDay: Integer;
    function GetMonth: Integer;
    function GetYear: Integer;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
  end;
```

Each of the Get and Set methods is easily implemented using functions available in the new DateUtils unit (discuss in more detail in Chapter 4). Here is the code for two of them (the others are very similar):

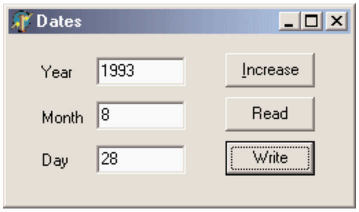
```
function TDate.GetYear: Integer;
begin
  Result := YearOf (fDate);
end;

procedure TDate.SetYear(const Value: Integer);
begin
  fDate := RecodeYear (fDate, Value);
end;
```

The code for this class is available in the DateProp example. The program uses a secondary unit for the definition of the TDate class to enforce encapsulation and creates a single-date object stored in a form variable and kept in memory for the entire execution of the program. Using a standard approach, the object is created in the form OnCreate event handler and destroyed in the form OnDestroy event handler.

The form of the program (see Figure 2.2) has three edit boxes and buttons to copy the values of these edit boxes to and from the properties of the date object:

FIGURE 2.2:

The form of the DateProp example



Year: 1993
Month: 8
Day: 28

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;
```

WARNING When writing the values, the program uses the `SetValue` method instead of setting each of the properties. In fact, assigning the month and the day separately can cause you trouble when the month is not valid for the current day. For example, the day is currently January 31, and you want to assign to it February 20. If you assign the month first, this part of the assignment will fail, as February 31 does not exist. If you assign the day first, the problem will arise when doing the reverse assignment. Due to the validity rules for dates, it is better to assign everything at once.

Advanced Features of Properties

Properties have several advanced features I'll focus on in future chapters, specifically the introduction to the base classes of the library in Chapter 5 and writing custom Delphi components in Chapter 11, "Creating Components." This is a short summary of these more advanced features:

- The `write` directive of a property can be omitted, making it a *read-only* property. The compiler will issue an error if you try to change it. You can also omit the `read` directive and define a *write-only* property, but that doesn't make much sense and is used infrequently.
- The Delphi IDE gives special treatment to *design-time* properties, declared with the `published` access specifier and generally displayed in the Object Inspector for the selected component. More on the `published` keyword and its effect is in Chapter 5.
- The other properties, often called *run-time only* properties, are those declared with the `public` access specifier. These properties can be used in the program code.

- You can define *array-based* properties, which use the typical notation with square brackets to access an element of a list. The *string list–based* properties, such as the `Lines` of a list box, are a typical example of this group.
- Properties have special directives, including `stored` and `default`, which control the *component streaming system*, introduced in Chapter 5 and detailed in Chapter 11.

NOTE You can usually assign a value to a property or read it, and you can even use properties in expressions, but you cannot always pass a property as a parameter to a procedure or method. This is because a property is not a memory location, so it cannot be used as a `var` parameter; it cannot be passed by reference.

Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a class's field, you only need to change the code of some methods of that class and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.

Let me clarify this idea with an example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of one of the forms:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

This works but has two problems. First, the data is not connected to a specific instance of the form, but to the entire program. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

The second problem is that if you define the data as a global variable or as a public field of a form, you won't be able to modify its implementation in the future without affecting the code that uses the data. For example, if you only have to read the current value from other forms, you can declare the data as private and provide a method to read the value:

```
type
  TForm1 = class(TForm)
  public
    function GetClicks: Integer;
  private
    nClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := nClicks;
end;
```

Adding Properties to Forms

An even better solution is to add a property to the form. Every time you want to make some information of a form available to other forms, you should really use a property, for all the reasons discussed in the previous section. Simply change the field declaration of the form, shown in the preceding listing, by adding the keyword `property` in front of it and then press Ctrl+Shift+C to activate code completion. Delphi will automatically generate all of the extra code you need. In the form, you also need to handle the `OnClick` event, increasing the value of the property (and showing it in the form caption):

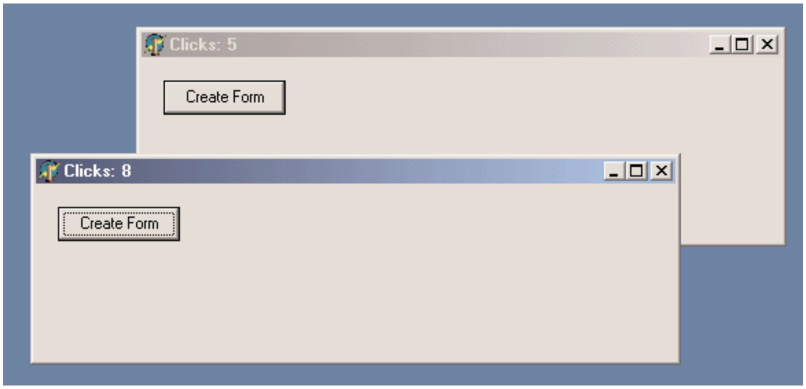
```
procedure TForm1.FormClick(Sender: TObject);
begin
  Inc (FClicks);
  Caption := 'Clicks: ' + IntToStr (FClicks);
end;
```

The complete code for this form class is available in the FormProp example and illustrated in Figure 2.3. The program can create multi-instances of the form (that is, multiple objects based on the same form class), each with its own click count. Clicking the Create Form button creates the secondary forms, using the following code:

```
procedure TForm1.btnCreateFormClick(Sender: TObject);
begin
  with TForm1.Create (Self) do
    Show;
end;
```

FIGURE 2.3:

Two forms of the FormProp example at run time



NOTE Notice that adding a property to a form doesn't add to the list of the form properties in the Object Inspector.

In my opinion, properties should also be used in the form classes to encapsulate the access to the components of a form. For example, if you have a main form with a status bar used to display some information (and with the `SimplePanel` property set to `True`) and you want to modify the text from a secondary form, you might be tempted to write:

```
Form1.StatusBar1.SimpleText := 'new text';
```

This is a standard practice in Delphi, but it's not a good one, because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (replacing `StatusBar` with another control or activating multiple panels), you'll have to fix the code in many places. The alternative is to use a method or, even better, a property to hide the specific control. Simply type

```
property StatusText: string read GetText write SetText;
```

and press the Ctrl+Shift+C combination again, to let Delphi add the definition of both methods for reading and writing the property:

```
function TForm1.GetText: string;
begin
  Result := StatusBar1.SimpleText;
end;

procedure TForm1.SetText(const Value: string);
begin
  StatusBar1.SimpleText := Value;
end;
```

In the other forms of the program, you can simply refer to the StatusText property of the form, and if the user interface changes, only the Set and Get methods of the property are affected.

NOTE See Chapter 5 for a detailed discussion of how you can avoid having published form fields for components, which will improve encapsulation. But don't rush there: the description requires a good knowledge of Delphi, and the technique discussed has a few drawbacks!

Constructors

As I've mentioned, to allocate the memory for the object, we call the Create method. This is a *constructor*, a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on. The default TObject.Create constructor initializes all the data of the new instance to zero.

If you want your instance data to start out with a nonzero value, then you need to write a custom constructor to do that. The new constructor can be called Create, or it can have any other name; use the constructor keyword in front of it. Notice that you don't need to call TObject.Create: it is Delphi that allocates the memory for the new object, not the class constructor. All you have to do is to initialize the class base.

If you create objects without initializing them, calling methods later may result in odd behavior or even a run-time error. A consistent use of constructors to initialize objects' data is an important *preventive* technique to avoid these errors in the first place. For example, we must call the SetValue procedure of the TDate class after we've created the object. As an alternative, we can provide a customized constructor, which creates the object and gives it an initial value.

Although you can use any name for a constructor, you should stick to the standard name, Create. If you use a name other than Create, the Create constructor of the base TObject class will still be available, but a programmer calling this default constructor might bypass the initialization code you've provided because they don't recognize the name.

By defining a Create constructor with some parameters, you replace the default definition with a new one and make its use compulsory. For example, after you define

```
type
  TDate = class
  public
    constructor Create (y, m, d: Integer);
constructor TDate.Create (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;
```

you'll be able to call this constructor and not the standard `Create`:

```
var
  ADay: TDate;
begin
  // Error, does not compile:
  ADay := TDate.Create;
  // OK:
  ADay := TDate.Create (1, 1, 2000);
```

The rules for writing constructors for custom components are different, as we'll see in Chapter 11. In short, when you inherit from `TComponent`, you should override the default `Create` constructor with one parameter and avoid disabling it.

Overloaded Constructors

Overloading is particularly relevant for constructors, because we can add to a class multiple constructors and call them all `Create`, which makes them easy to remember.

NOTE Historically, overloading was added to C++ to allow the use of multiple constructors that have the same name (the name of the class). In Object Pascal, this feature was considered unnecessary because multiple constructors can have different specific names. The increased integration of Delphi with C++Builder has motivated Borland to make this feature available in both languages, starting with Delphi 4. Technically, when C++Builder constructs an instance of a Delphi VCL class, it looks for a Delphi constructor named `Create` and nothing but `Create`. If the Delphi class has constructors by other names, they cannot be used from C++Builder code. Therefore, when creating classes and components you intend to share with C++Builder programmers, you should be careful to name all your constructors `Create` and distinguish between them by their parameter lists (using `overload`). Delphi does not require this, but it is required for C++Builder to use your Delphi classes.

As an example, I've added to the class two separate `Create` constructors: one with no parameters, which hides the default constructor, and one with the initialization values. The constructor with no parameter uses as the default value today's date:

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
constructor TDate.Create (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

constructor TDate.Create;
begin
  fDate := Date;
end;
```

Having these two constructors makes it possible to define a new `TDate` object in two different ways:

```
var
  Day1, Day2: TDate;
begin
  Day1 := TDate.Create (2001, 12, 25);
  Day2 := TDate.Create; // today
```

See the section “The Complete *TDate* Class” later in this chapter for the DateView example, which includes the code of these constructors.

Destructors

In the same way that a class can have a custom constructor, it can have a custom destructor, a method declared with the `destructor` keyword and called `Destroy`, which can perform some resource cleanup before an object is destroyed. Just as a constructor call allocates memory for the object, a destructor call frees the memory.

We can write code for a destructor, generally overriding the default `Destroy` destructor, to let the object execute some code before it is destroyed. Destructors are needed only for objects that acquire resources in their constructors or during their lifetime. In your code, of course, you don’t have to handle memory de-allocation—this is something Delphi does for you.

`Destroy` is a virtual destructor of the `TObject` class. Most of the classes that require custom clean-up code when the objects are destroyed override this virtual method. The reason you should never define a new destructor is that objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor of the specific class (virtual methods will be discussed later in this chapter).

Free (and *nil*)

`Free` is a method of the `TObject` class, inherited by all other classes. The `Free` method basically checks whether the current object (`Self`) is not `nil` before calling the `Destroy` virtual destructor. Here is its pseudocode (the actual Delphi code is written in assembler):

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

By looking at this code, you can see that calling `Free` doesn't set the object to `nil` automatically; this is something you should do yourself! The reason is that the object doesn't know which variables may be referring to it, so it has no way to set them all to `nil`.

NOTE Automatically setting an object to `nil` is not possible. You might have several references to the same object, and Delphi doesn't track them. At the same time, within a method (such as `Free`) we can operate on the object, but we know nothing about the object reference—the memory address of the variable we've used to call the method. In other words, inside the `Free` method or any other method of a class, we know the memory address of the object (`Self`), but we don't know the memory location of the variable referring to the object.

Delphi 5 introduced a `FreeAndNil` procedure you can use to free an object and set its reference to `nil` at the same time. Simply call

```
FreeAndNil (Obj1)
```

instead of writing

```
Obj1.Free;
Obj1 := nil;
```

The `FreeAndNil` procedure knows about the object reference, passed as a parameter, and can act on it. Here is Delphi code for `FreeAndNil`:

```
procedure FreeAndNil(var Obj);
var
  P: TObject;
begin
  P := TObject(Obj);
  // clear the reference before destroying the object
  TObject(Obj) := nil;
  P.Free;
end;
```

NOTE There's more on this topic in the section "Destroying Objects Only Once" later in this chapter.

The Complete *TDate* Class

In the initial portion of this chapter, I've shown you bits and pieces of the source code for different versions of a TDate class. In Listing 2.1 is the complete interface portion of the unit that defines the TDate class.

Listing 2.1: The *TDate* class, from the ViewDate example

```
unit Dates;

interface

type
  TDate = class
  private
    fDate: TDateTime;
    procedure SetDay(const Value: Integer);
    procedure SetMonth(const Value: Integer);
    procedure SetYear(const Value: Integer);
    function GetDay: Integer;
    function GetMonth: Integer;
    function GetYear: Integer;
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string;
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
  end;

implementation

uses
  SysUtils, DateUtils;

procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

function TDate.LeapYear: Boolean;
begin
  Result := IsInLeapYear(fDate);
end;
```

```
procedure TDate.Increase (NumberOfDays: Integer = 1);
begin
  fDate := fDate + NumberOfDays;
end;

function TDate.GetText: string;
begin
  GetText := DateToStr (fDate);
end;

procedure TDate.Decrease (NumberOfDays: Integer = 1);
begin
  fDate := fDate - NumberOfDays;
end;

constructor TDate.Create (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

constructor TDate.Create;
begin
  fDate := Date;
end;

procedure TDate.SetValue(NewDate: TDateTime);
begin
  fDate := NewDate;
end;

procedure TDate.SetDay(const Value: Integer);
begin
  fDate := RecodeDay (fDate, Value);
end;

procedure TDate.SetMonth(const Value: Integer);
begin
  fDate := RecodeMonth (fDate, Value);
end;

procedure TDate.SetYear(const Value: Integer);
begin
  fDate := RecodeYear (fDate, Value);
end;

function TDate.GetDay: Integer;
begin
  Result := DayOf (fDate);
end;
```

```
function TDate.GetMonth: Integer;
begin
  Result := MonthOf (fDate);
end;

function TDate.GetYear: Integer;
begin
  Result := YearOf (fDate);
end;

end.
```

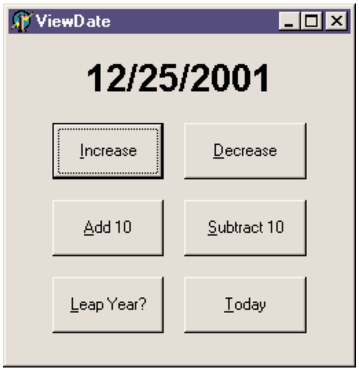
The aim of the Increase and Decrease methods, which have a default value for their parameter, is quite easy to understand. If called with no parameter, they change the value of the date to the next or previous day. If a NumberOfDays parameter is part of the call, they add or subtract that number.

GetText returns a string with the formatted date, using the DateToStr function.

The form of the example I've built to show you how to use the TDate class, as illustrated in Figure 2.4, has a caption to display a date and six buttons, which can be used to modify the date. To make the label component look nice, I've given it a big font, made it as wide as the form, set its Alignment property to taCenter, and set its AutoSize property to False.

FIGURE 2.4:

The output of the ViewDate example at startup



The startup code of this program is in the OnCreate event handler. In the corresponding method, we create an instance of the TDate class, initialize this object, and then show its textual description in the Caption of the label.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Create (2001, 12, 25);
  LabelDate.Caption := TheDay.GetText;
end;
```

TheDay is a private field of the class of the form, TDateForm. (By the way, the name for the form class is automatically chosen by Delphi when we change the Name property of the form to DateForm.) The object is then destroyed along with the form:

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
  TheDay.Free;
end;
```

When the user clicks one of the six buttons, we need to apply the corresponding method to the TheDay object and then display the new value of the date in the label:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  TheDay.SetValue (Date);
  LabelDate.Caption := TheDay.GetText;
end;
```

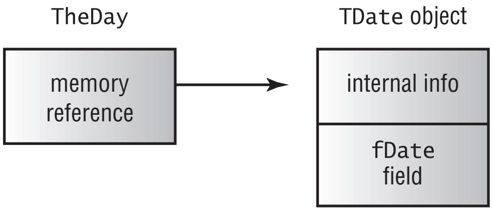
Notice that in this code we reuse an existing object, assigning a new date to it. We could also create a new object and assign it to the existing TheDate variable, but this can lead to confusing situations, as explained in the next section.

Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Object Pascal, instead, is based on an *object reference model*. The idea is that a variable of a class type, such as the TheDay variable in the preceding ViewDate example, does not hold the value of the object. Rather, it contains a reference, or a *pointer*, to indicate the memory location where the object has been stored. You can see this structure depicted in Figure 2.5.

FIGURE 2.5:

A representation of the structure of an object in memory, with a variable referring to it



The only problem with this approach is that when you declare a variable, you don't create an object in memory; you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of the components you place on a form are built automatically by Delphi.

You've seen how to create an instance of an object by applying a constructor to its class. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the Free method. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch. The object reference model has many consequences on assigning object and on managing memory, as we'll see in the next two sections.

Assigning Objects

If a variable holding an object only contains a reference to the object in memory, what happens if you copy the value of that variable? Suppose we write the BtnTodayClick method of the ViewDate example in the following way:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Create;
  TheDay := NewDay;
  LabelDate.Caption := TheDay.GetText;
end;
```

This code copies the memory address of the NewDay object to the TheDay variable (as shown in Figure 2.6); it doesn't copy the data of an object into the other. In this particular circumstance, this is not a very good approach, as we keep allocating memory for a new object every time the button is pressed, but we never release the memory of the object the TheDay variable was previously pointing to. This specific issue can be solved by freeing the old object, as in the following code (which is also simplified, without the use of an explicit variable for the newly created object):

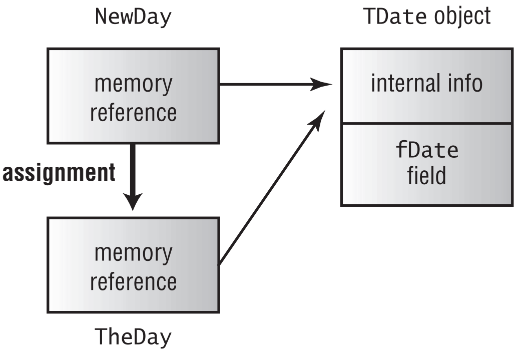


FIGURE 2.6:

A representation of the operation of assigning an object reference to another one. This is different from copying the actual content of an object to another.

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  TheDay.Free;
  TheDay := TDate.Create;
```

The important thing to keep in mind is that, when you assign an object to another object, Delphi copies the reference to the object in memory to the new object/reference. You should not consider this a negative: In many cases, being able to define a variable referring to an existing object can be a plus. For example, you can store the object returned by calling a function or accessing a property and use it in subsequent statements, as this code snippet indicates:

```
var
  ADay: TDate;
begin
  ADay: UserInformation.GetBirthDate;
  // use a ADay
```

The same happens if you pass an object as a parameter to a function: You don't create a new object, but you refer to the same one in two different places of the code. For example, by writing this procedure and calling it as follows, you'll modify the `Caption` property of the `Button1` object, not of a copy of its data in memory (which would be totally useless):

```
procedure CaptionPlus (Button: TButton);
begin
  Button.Caption := Button.Caption + '+';
end;

// call...
CaptionPlus (Button1)
```

What if you really want to change the data inside an existing object, so that it matches the data of another object? You have to copy each field of the object, which is possible only if they are all public, or provide a specific method to copy the internal data. Some classes of the VCL have an `Assign` method, which does this copy operation. To be more precise, most of the VCL classes inheriting from `TPersistent`, but not inheriting from `TComponent`, have the `Assign` method. Other `TComponent`-derived classes have this method but raise an exception when it is called.

In the DateCopy example, slightly modified from the ViewDate program, I've added an `Assign` method to the `TDate` class, and I've called it from the Today button, with the following code:

```
procedure TDate.Assign (Source: TDate);
begin
  fDate := Source.fDate;
end;
```

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Create;
  TheDay.Assign(NewDay);
  LabelDate.Caption := TheDay.GetText;
  NewDay.Free;
end;
```

Objects and Memory

Memory management in Delphi is subject to three rules: Every object must be created before it can be used; every object must be destroyed after it has been used; and every object must be destroyed only once. Whether you have to do these operations in your code, or you can let Delphi handle memory management for you, depends on the model you choose among the different approaches provided by Delphi.

Delphi supports three types of memory management for dynamic elements (that is, elements not in the stack and the global memory area):

- Every time you create an object explicitly, in the code of your application, you should also free it. If you fail to do so, the memory used by that object won't be released for other objects until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form) becomes responsible for destroying all the objects it owns. In other words, when you free the form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it. This is the standard behavior of the components you create at design time by placing them on a form or data module.
- When you allocate memory for strings, dynamic arrays, and objects referenced by interface variables (discussed in Chapter 3), Delphi automatically frees the memory when the reference goes out of scope. You don't need to free a string: when it becomes unreachable, its memory is released.

Destroying Objects Only Once

Another problem is that if you call the `Destroy` destructor of an object twice, you get an error. If you remember to set the object to `nil`, you can call `Free` twice with no problem.

NOTE You might wonder why you can safely call Free if the object reference is nil, but you can't call Destroy. The reason is that Free is a known method at a given memory location, whereas the virtual function Destroy is determined at run time by looking at the type of the object, a very dangerous operation if the object doesn't exist any more.

To sum things up, here are a couple of guidelines:

- Always call Free to destroy objects, instead of calling the Destroy destructor.
- Use FreeAndNil, or set object references to nil after calling Free, unless the reference is going out of scope immediately afterward.

In general, you can also check whether an object is nil by using the Assigned function. So the following two statements are equivalent, at least in most cases:

```
if Assigned (ADate) then ...
if ADate <> nil then ...
```

Notice that these statements test only whether the pointer is not nil; they do not check whether it is a valid pointer. If you write the following code, the test will be satisfied, and you'll get an error on the line with the call to the method of the object:

```
ToDestroy.Free;
if ToDestroy <> nil then
  ToDestroy.DoSomething;
```

It is important to realize that calling Free doesn't set the object to nil.

What's Next?

In this chapter, we have discussed the foundations of object-oriented programming (OOP) in Object Pascal. We have considered the definition of classes, the use of methods, encapsulation, and memory management, but also some more advanced concepts such as properties and the dynamic creation of components.

This is certainly a lot of information if you are a newcomer, but if you are fluent in another OOP language or if you've already used past versions of Delphi, you should be able to apply the topics covered in this chapter to your programming.

The next chapter continues on the same line, highlighting inheritance in particular, along with virtual functions and interfaces. It also includes a discussion on exception handling and class references, so that at the end you'll have a complete overview of the language. At that point, you'll be ready to start focusing on the libraries the compiler relies on, and we'll get back to see how properties are used by Delphi and its IDE (in Chapter 5). Other chapters will provide further information on applying the OOP concepts to Delphi programming. You'll find OOP tips throughout the entire book, but particularly in Chapter 11, devoted to writing custom Delphi components.

CHAPTER 3

The Object Pascal Language: Inheritance and Polymorphism

- Inheritance
- Virtual methods
- Polymorphism
- Type-safe down-casting (run-time type information)
- Interfaces
- Working with exceptions
- Class references

After the introduction to classes and objects we've seen over the last chapter, let's move on to another key element of the language, *inheritance*. Deriving a class from an existing one is the real revolutionary idea of object-oriented programming, and it goes along with polymorphism, virtual functions, abstract functions, and many other topics discussed in this chapter.

We'll focus also on interfaces, another intriguing idea of the most recent OOP languages, and we'll cover a few more elements of Object Pascal, such as exception handling and class references. Together with the previous chapter, this will provide an almost complete roundup of the language.

Inheriting from Existing Types

We often need to use a slightly different version of an existing class that we have written or that someone has given to us. For example, you might need to add a new method or slightly change an existing one. You can do this easily by modifying the original code, unless you want to be able to use the two different versions of the class in different circumstances. Also, if the class was originally written by someone else (including Borland), you might want to keep your changes separate.

A typical alternative is to make a copy of the original type definition, change its code to support the new features, and give a new name to the resulting class. This might work, but it also might create problems: In duplicating the code you also duplicate the bugs; and if you want to add a new feature, you'll need to add it two or more times, depending on the number of copies of the original code you've made. This approach results in two completely different data types, so the compiler cannot help you take advantage of the similarities between the two types.

To solve these kinds of problems in expressing similarities between classes, Object Pascal allows you to define a new class directly from an existing one. This technique is known as *inheritance* (or *subclassing*) and is one of the fundamental elements of object-oriented programming languages. To inherit from an existing class, you only need to indicate that class at the beginning of the declaration of the subclass. For example, Delphi does this automatically each time you create a new form:

```
type
  TForm1 = class(TForm)
  end;
```

This simple definition indicates that the `TForm1` class inherits all the methods, fields, properties, and events of the `TForm` class. You can apply any public method of the `TForm` class to an object of the `TForm1` type. `TForm`, in turn, inherits some of its methods from another class, and so on, up to the `TObject` base class.

As an example of inheritance, we can change the ViewDate program, deriving a new class from TDate and modifying its GetText function. You can find this code in the DATES.PAS file of the NewDate example on the companion CD.

```
type
  TNewDate = class (TDate)
  public
    function GetText: string;
  end;
```

In this example, the TNewDate class is derived from TDate. It is common to say that TDate is an *ancestor* class or *parent* class of TNewDate and that TNewDate is a *subclass*, *descendant* class, or *child* class of TDate.

To implement the new version of the GetText function, I used the FormatDateTime function, which uses (among other features) the predefined month names available in Windows; these names depend on the user's regional and language settings. Many of these regional settings are actually copied by Delphi into constants defined in the library, such as LongMonthNames, ShortMonthNames, and many others you can find under the "Currency and date/time formatting variables" topic in the Delphi Help file. Here is the GetText method, where '*dddddd*' stands for the long date format:

```
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

TIP Using regional information, the NewDate program automatically adapts itself to different Windows user settings. If you run this same program on a computer with regional settings referring to a language other than English, it will automatically show month names in that language. To test this behavior, you just need to change the regional settings; you don't need a new version of Windows. Notice that regional-setting changes immediately affect the running programs.

Once we have defined the new class, we need to use this new data type in the code of the form of the NewDate example. Simply define the TheDay object of type TNewDate, and call its constructor in the FormCreate method:

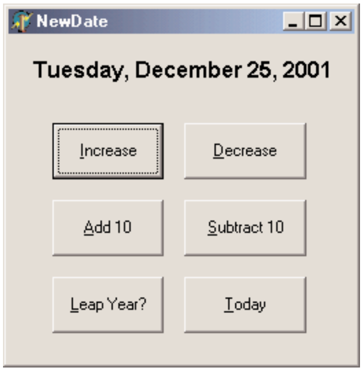
```
type
  TDateForm = class(TForm)
    ...
  private
    TheDay: TNewDate; // updated declaration
  end;
```

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TNewDate.Create (2001, 12, 25); // updated
  DateLabel.Caption := TheDay.GetText;
end;
```

Without any other changes, the new NewDate example will work properly. The TNewDate class inherits the methods to increase the date, add a number of days, and so on. In addition, the older code calling these methods still works. Actually, to call the new version of the GetText method, we don't need to change the source code! The Delphi compiler will automatically bind that call to a new method. The source code of all the other event handlers remains exactly the same, although its meaning changes considerably, as the new output demonstrates (see Figure 3.1).

FIGURE 3.1:

The output of the NewDate program, with the name of the month and of the day depending on Windows regional settings



Tuesday, December 25, 2001

Protected Fields and Encapsulation

The code of the GetText method of the TNewDate class compiles only if it is written in the same unit as the TDate class. In fact, it accesses the fDate private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the fDate field as protected or add a protected access method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write subclasses. However, this violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If ten derived classes are accessing this data, changing its definition means potentially modifying the code in each of the ten classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate solution can be obtained by using a virtual method, a topic I'll discuss in detail later in the section "Late Binding and Polymorphism." If you choose not to use encapsulation in order to obtain faster coding of the subclasses, then your design might not follow the object-oriented principles.

Accessing Protected Data of Other Classes

We've seen that in Delphi, the `private` and `protected` data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this class (part of the Protection example on the companion CD):

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  public
    PublicData: Integer;
    function GetValue: string;
  end;
```

The `GetValue` method simply returns a string with the two integer values:

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

Once you place this class in its own unit, you won't be able to access its protected portion from other units directly. Accordingly, if you write the following code,

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20;  // won't compile
  ShowMessage (Obj.GetValue);
  Obj.Free;
end;
```

the compiler will issue an error message, "Undeclared identifier: 'ProtectedData.'"

Continued on next page

At this point, you might think there is no way to access the protected data of a class defined in a different unit. (This is what Delphi manuals and most Delphi books say.) However, there is a way around it. Consider what happens if you create an apparently useless derived class, such as

```
type
  TFake = class (TTest);
```

Now, if you make a direct cast of the object to the new class and access the protected data through it, this is how the code will look:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; // compiles!
  ShowMessage (Obj.GetValue);
  Obj.Free;
end;
```

This code compiles and works properly, as you can see by running the Protection program. How is it possible for this approach to work? Well, if you think about it, the `TFake` class automatically inherits the protected fields of the `TTest` base class, and because the `TFake` class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible. As you would expect, if you move the declaration of the `TFake` class to a secondary unit, the program won't compile any more.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are times when using this technique is the best solution, as you'll see by looking at the VCL source code and the code of many Delphi components. Two examples that come to mind are accessing the `Text` property of the `TControl` class and the `Row` and `Col` positions of the `DBGrid` control. These two ideas are demonstrated by the TextProp and DBGridCol examples, respectively. (These examples are quite advanced, so I suggest that only programmers with a good background of Delphi programming read them at this point in the text—other readers might come back later.) Although the first example shows a reasonable example of using the typecast *cracker,* the DBGrid example of `Row` and `Col` is actually a counterexample, one that illustrates the risks of accessing bits that the class writer chose not to expose. The row and column of a `DBGrid` do not mean the same thing as they do in a `DrawGrid` or `StringGrid` (the base classes). First, `DBGrid` does not count the fixed cells as actual cells (it distinguishes data cells

Continued on next page

from decoration), so your row and column indexes will have to be adjusted by whatever decorations are currently in effect on the grid (and those can change on the fly). Second, the `DBGrid` is a virtual view of the data. When you scroll up in a `DBGrid`, the data may move underneath it, but the currently selected row might not change.

This technique—declaring a local type only so that you can access protected data members of a class—is often described as a *hack,* and it should be avoided whenever possible. The problem is not accessing protected data of a class in the same unit but declaring a class for the sole purpose of accessing protected data of an existing object of a different class! The danger of this technique is in the hard-coded typecast of an object from a class to a different one.

Inheritance and Type Compatibility

Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a `Boolean` variable, unless you use an explicit typecast. The rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type refers to a single type definition.

WARNING If you redefine the same data type in two different units, they won't be compatible, even if their name is identical. A program using two equally named types of two different units will be a nightmare to compile and debug.

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. That is because a dog is an animal! So, although this might surprise you, the following constructor calls are both legal:

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

As a general rule, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
type
  TDog = class (TAnimal)
    ...
  end;
```

```
var
  MyAnimal: TAnimal;
  MyDog: TDog;

begin
  MyAnimal := MyDog;  // This is OK
  MyDog := MyAnimal;  // This is an error!!!
```

Before we look at the implications of this important feature of the language, you can try out the Animals1 example from the companion CD, which defines the two `TAnimal` and `TDog` classes:

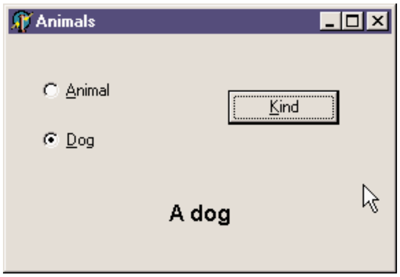
```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

The two `Create` methods set the value of `Kind`, which is returned by the `GetKind` function. The form displayed by this example, shown in Figure 3.2, has a private field `MyAnimal` of type `TAnimal`. An instance of this class is created and initialized when the form is created and each time one of the radio buttons is selected:

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioDogClick(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

FIGURE 3.2:

The form of the Animals1 example



Finally, the Kind button calls the `GetKind` method for the current animal and displays the result in the label:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  KindLabel.Caption := MyAnimal.GetKind;
end;
```

Late Binding and Polymorphism

Pascal functions and procedures are usually based on *static* or *early binding*. This means that a method call is resolved by the compiler and linker, which replace the request with a call to the specific memory location where the function or procedure resides. (This is known as the *address* of the function.) OOP languages allow the use of another form of binding, known as *dynamic* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

The advantage of this technique is known as *polymorphism*. Polymorphism means you can write a call to a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the actual class of the object the variable refers to, because of the type-compatibility rule discussed in the previous section.

NOTE The term *polymorphism* is quite a mouthful. A glance at the dictionary tells us that in a general sense, it refers to something having more than one form. In the OOP sense, then, it refers to the facts that there may be several versions of a given method across several related classes and that a single method call on an object instance of a particular class type can refer to one of these versions. Which version of the method gets called depends on the type of the object instance used to make the call at run time.

For example, suppose that a class and its subclass (let's say TAnimal and TDog) both define a method, and this method has late binding. Now you can apply this method to a generic variable, such as MyAnimal, which at run time can refer either to an object of class TAnimal or to an object of class TDog. The actual method to call is determined at run time, depending on the class of the current object.

The Animals2 example extends the Animals1 program to demonstrate this technique. In the new version, the TAnimal and the TDog classes have a new method: Voice, which means to output the sound made by the selected animal, both as text and as sound. This method is defined as virtual in the TAnimal class and is later overridden when we define the TDog class, by the use of the virtual and override keywords:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

Of course, the two methods also need to be implemented. Here is a simple approach:

```
uses
  MMSystem;

function TAnimal.Voice: string;
begin
  Voice := 'Voice of the animal';
  PlaySound ('Anim.wav', 0, snd_Async);
end;

function TDog.Voice: string;
begin
  Voice := 'Arf Arf';
  PlaySound ('dog.wav', 0, snd_Async);
end;
```

TIP This example uses a call to the PlaySound API function, defined in the MMSystem unit. The first parameter of this function is the name of the WAV sound file or the system sound you want to execute. The second parameter indicates an optional resource file containing the sound. The third parameter indicates (among other options) whether the call should be synchronous or asynchronous; that is, whether the program should wait for the sound to finish before continuing with the following statements.

Now what is the effect of the call `MyAnimal.Voice`? It depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice` instead. This happens only because the function is `virtual` (as you can experiment by removing this keyword and recompiling).

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined in other units—or that haven't been written yet! The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` subclasses.

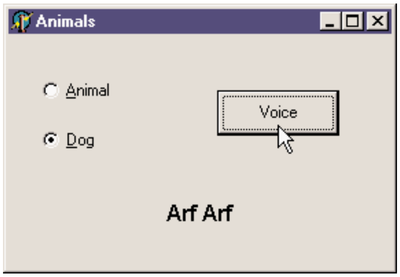
NOTE This is the key technical reason why object-oriented programming languages favor reusability. You can write code that uses classes within a hierarchy without any knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition: the ancestor classes of the hierarchy need to be designed very carefully.

The Animals2 program demonstrates the use of these new classes and has a form similar to that of the previous example. This code is executed by clicking the button:

```
procedure TFormAnimals.BtnVerseClick(Sender: TObject);
begin
  LabelVoice.Caption := MyAnimal.Voice;
end;
```

In Figure 3.3, you can see an example of the output of this program. By running it, you'll also hear the corresponding sounds produced by the `PlaySound` API call.

FIGURE 3.3: The output of the Animals2 example



Overriding and Redefining Methods

As we have just seen, to override a late-bound method in a descendant class, you need to use the override keyword. Note that this can take place only if the method was defined as virtual in the ancestor class. Otherwise, if it was a static method, there is no way to activate late binding, other than by changing the code of the ancestor class.

The rules are simple: A method defined as static remains static in every subclass, unless you hide it with a new virtual method having the same name. A method defined as virtual remains late-bound in every subclass. There is no way to change this, because of the way the compiler generates different code for late-bound methods.

NOTE The new C# programming language proposed by Microsoft (which is in essence a clone of Java) has the same notion as the Object Pascal language of marking the overridden version of a method with a specific keyword.

To redefine a static method, you add a method to a subclass having the same parameters or different parameters than the original one, without any further specifications. To override a virtual method, you must specify the same parameters and use the override keyword:

```
type
  MyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;

  MySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

There are typically two ways to override a method. One is to replace the method of the ancestor class with a new version. The other is to add some more code to the existing method. This can be accomplished by using the inherited keyword to call the same method of the ancestor class. For example, you can write

```
procedure MySubClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

You might wonder why you need to use the override keyword. In other languages, when you redefine a method in a subclass, you automatically override the original one. However, having a specific keyword allows the compiler to check the correspondence between the names of the methods of the ancestor class and the subclass (misspelling a redefined function is a common error in other OOP languages), check that the method was virtual in the ancestor class, and so on.

When you override an existing virtual method of a base class, you must use the same parameters. When you introduce a new version of a method in a descendent class, you can declare it with the parameters you want. In fact, this will be a new method unrelated to the ancestor method of the same name. They only happen to use the same name. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string);
  end;
```

NOTE Using the class definitions above, when you create an object of the `TMySubClass` class, you can apply to it the `One` method with the string parameter, but not the parameter-less version defined in the base class. If this is what you need, it can be accomplished by marking the re-declared method (the one in the derived class) with the `overload` keyword. If the method has different parameters than the version in the base class, it becomes effectively an overloaded method; otherwise it replaces the base class method. Notice that the method doesn't need to be marked as `overload` in the base class. However, if the method in the base class is virtual, the compiler issues the warning "Method 'One' hides virtual method of base type 'TMyClass.'" To avoid this message and to instruct the compiler more precisely on your intentions, you can use the `reintroduce` directive. If you are interested in this advanced topic, you can find this code in the Reintr example on the companion CD and experiment with it further.

Virtual versus Dynamic Methods

In Delphi, there are two different ways to activate late binding. You can declare the method as `virtual`, as we have seen before, or declare it as `dynamic`. The syntax of these two keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

`virtual` methods are based on a *virtual method table* (VMT, also known as a *vtable*), which is an array of method addresses. For a call to a `virtual` method, the compiler generates code to jump to an address stored in the *n*th slot in the object's virtual method table.

Virtual method tables allow fast execution of the method calls. Their main drawback is that they require an entry for each `virtual` method for each descendant class, even if the method is not overridden in the subclass. At times, this has the effect of propagating VMT entries throughout a class hierarchy (even for methods that aren't redefined). This might require a lot of memory just to store the same method address multiple times.

`Dynamic` method calls, on the other hand, are dispatched using a unique number indicating the method. The search for the corresponding function is generally slower than the one-step table lookup for `virtual` methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method. For large or deep object hierarchies, using `dynamic` methods instead of `virtual` methods can result in significant memory savings with only a minimal speed penalty.

From a programmer's perspective, the difference between these two approaches lies only in a different internal representation and slightly different speed or memory usage. Apart from this, `virtual` and `dynamic` methods are the same.

Message Handlers

A late-bound method can be used to handle a Windows message, too, although the technique is somewhat different. For this purpose Delphi provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter. The `message` directive is followed by the number of the Windows message the method wants to handle.

WARNING The `message` directive is also available in Delphi for Linux and is fully supported by the language and the RTL. However, the visual portion of the CLX application framework does not use message methods to dispatch notifications to controls. For this reason, whenever possible, you should use a virtual method provided by the library rather than handle a Windows message directly. Of course, this matters only if you want your code to be more portable.

For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `wm_User` Windows constant:

```
type
  TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage);
      message wm_User;
  end;
```

The name of the procedure and the actual type of the parameters are up to you, although there are several predefined record types for the various Windows messages. You could later send this message, invoking the corresponding method, by writing:

```
PostMessage (Form1.Handle, wm_User, 0, 0);
```

This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions. You can also dispatch a message to an object by calling the TObject.Dispatch method on the object. This will be a synchronous message call, not asynchronous like PostMessage. TObject.Dispatch is fully platform independent.

The ability to handle Windows messages and call API functions as you do when you are programming Windows with the C language may horrify some programmers and delight others. But in Delphi, when writing Windows applications, you will seldom need to use message methods or call Windows APIs directly. Obviously, these techniques will also affect the portability of your code to other platforms.

Abstract Methods

The abstract keyword is used to declare methods that will be defined only in subclasses of the current class. The abstract directive fully defines the method; it is not a forward declaration. If you try to provide a definition for the method, the compiler will complain. In Object Pascal, you can create instances of classes that have abstract methods. However, when you try to do so, Delphi's 32-bit compiler issues the warning message "Constructing instance of <class name> containing abstract methods." If you happen to call an abstract method at run time, Delphi will raise an exception, as demonstrated by the following Animals3 example.

NOTE C++ and Java use a more strict approach: in these languages, you cannot create instances of classes containing abstract methods.

You might wonder why you would want to use abstract methods. The reason lies in the use of polymorphism. If class TAnimal has the abstract method Voice, every subclass can redefine it. The advantage is that you can now use the generic MyAnimal object to refer to each animal defined by a subclass and invoke this method. If this method was not present in the interface of the TAnimal class, the call would not have been allowed by the compiler, which performs static type checking. Using a generic MyAnimal object, you can call only the method defined by its own class, TAnimal.

You cannot call methods provided by subclasses, unless the parent class has at least the declaration of this method—in the form of an abstract method. The next example, Animals3, demonstrates the use of abstract methods and the abstract call error. In Listing 3.1, you can see the interfaces of the classes of this new example. (Here TAnimal is an abstract class.)

Listing 3.1: Declaration of the three classes of the Animals3 example

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Voice: string; virtual; abstract;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;

  TCat = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;
```

The most interesting portion of Listing 3.1 is the definition of the class TAnimal, which includes a virtual abstract method: Voice. It is also important to notice that each derived class overrides this definition and adds a new virtual method, Eat. What are the implications of these two different approaches? To call the Voice function, we can write the same code as in the previous version of the program:

```
LabelVoice.Caption := MyAnimal.Voice;
```

How can we call the Eat method? We cannot apply it to an object of the TAnimal class. The statement

```
LabelVoice.Caption := MyAnimal.Eat;
```

generates the compiler error “Field identifier expected.”

To solve this problem, you can use run-time type information (RTTI) to cast the TAnimal object to a TCat or TDog object; but without the proper cast, the program will raise an exception. You will see an example of this approach in the next section. Adding the method definition to the TAnimal class is a typical solution to the problem, and the presence of the abstract keyword favors this choice.

NOTE What happens if a method overriding an abstract method calls `inherited`? In past versions of Delphi, this resulted in an abstract method call. In Delphi 6, the compiler has been enhanced to notice the presence of the abstract method and simply skip the `inherited` call. This means you can safely always use `inherited` in every overridden method, unless you specifically want to disable executing some code of the base class.

Type-Safe Down-Casting

The Object Pascal type-compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned earlier, the reverse is not possible.

Now suppose that the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, it should be possible to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, we could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods we are calling actually exist.

To solve the problem, we can use techniques based on *run-time type information* (RTTI, for short). Essentially, because each object “knows” its type and its parent class, and we can ask for this information with the `is` operator or using the `InheritsFrom` method of the `TObject` class. The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then ...
```

The `is` expression evaluates as `True` only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from `TDog`. This means that if you test whether a `TDog` object is of type `TAnimal`, the test will succeed. In other words, this expression evaluates as `True` if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

Now that you know for sure that the animal is a dog, you can make a safe typecast (or type conversion). You can accomplish this direct cast by writing the following code:

```
var
  MyDog: TDog;
begin
  if MyAnimal is TDog then
  begin
    MyDog := TDog (MyAnimal);
    Text := MyDog.Eat;
  end;
```

This same operation can be accomplished directly by the second RTTI operator, `as`, which converts the object only if the requested class is compatible with the actual one. The parameters of the `as` operator are an object and a class type, and the result is an object converted to the new class type. We can write the following snippet:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

If we only want to call the `Eat` function, we might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second raises an exception if the type of the object is incompatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described at the end of this chapter).

To avoid this exception, use the `is` operator and, if it succeeds, make a plain typecast (in fact, there is no reason to use `is` and `as` in sequence, doing the type check twice):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both RTTI operators are very useful in Delphi because you often want to write generic code that can be used with several components of the same type or even of different types. When a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`), so you often need to cast it back to the original component type:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

This is a common technique in Delphi, and I'll use it in examples throughout the book. The two RTTI operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism first. Only in special cases, where polymorphism alone cannot be applied, should you try using the RTTI operators to complement it. *Do not use RTTI instead of polymorphism.* This is bad programming practice, and it results in slower programs. RTTI, in fact, has a negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As we have seen, virtual method calls require just a memory lookup, which is much faster.

NOTE There is actually more to run-time type information (RTTI) than the `is` and `as` operators. You can access to detailed class and type information at run time, particularly for published properties, events, and methods. More on this topic in Chapter 5.

Using Interfaces

When you define an abstract class to represent the base class of a hierarchy, you can come to a point in which the abstract class is so abstract that it only lists a series of virtual functions without providing any actual implementation. This kind of *purely abstract class* can also be defined using a specific technique, an `interface`. For this reason, we refer to these classes as *interfaces*.

Technically, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element with distinctive features:

- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to how Delphi manages long strings and makes memory management almost automatic.
- A class can inherit from a single base class, but it can implement multiple interfaces.
- As all classes descend from `TObject`, all interfaces descend from `IInterface`, forming a totally separate hierarchy.

The base interface class used to be `IUnknown` until Delphi 5, but Delphi 6 introduces a new name for it, `IInterface`, to mark even more clearly the fact that this language feature is separate from Microsoft's COM. In fact, Delphi interfaces are available also in the Linux version of the product.

You can use this rule: Interface types describing things that relate to COM and the related operating-system services should inherit from `IUnknown`. Interface types that describe things that do not necessarily require COM (for example, interfaces used for the internal application structure) should inherit from `IInterface`. Doing this consistently in your applications will make it easier to identify which portions of your application probably assume or require the Windows operating system and which portions are probably OS-independent.

NOTE Borland introduced interfaces in Delphi 3 along with the support COM programming. Though the interface language syntax may have been created to support COM, interfaces do not require COM. You can use interfaces to implement abstraction layers within your applications, without building COM server objects. For example, the Delphi IDE uses interfaces extensively in its internal architecture. COM is discussed in Chapter 19.

From a more general point of view, interfaces support a slightly different object-oriented programming model than classes. Objects implementing interfaces are subject to polymorphism for each of the interfaces they support. Indeed, the interface-based model is powerful. But having said that, I'm not interested in trying to assess which approach is better in each case. Certainly, interfaces favor encapsulation and provide a looser connection between classes than inheritance. Notice that the most recent OOP languages, from Java to C#, have the notion of interfaces.

Here is the syntax of the declaration of an interface (which, by convention, starts with the letter *I*):

```
type
  ICanFly = interface
    ['{EAD9C4B4-E1C5-4CF4-9FA0-3B812C880A21}']
    function Fly: string;
  end;
```

The above interface has a GUID, a numeric ID following its declaration and based on Windows conventions. You can generate these identifiers (called GUIDs in jargon) by pressing Ctrl+Shift+G in the Delphi editor.

Although you can compile and use interfaces even without specifying a GUID (as in the code above) for them, you'll generally want to do it, as this is required to perform `QueryInterface` or dynamic `as` typecasts using that interface type. Since the whole point of interfaces is (usually) to take advantage of greatly extended type flexibility at run time, if compared with class types, interfaces without GUIDs are not very useful.

Once you've declared an interface, you can define a class to implement it, as in:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

The RTL already provides a few base classes to implement the basic behavior required by the `IInterface` interface. The simplest one is the `TInterfacedObject` class I've used in this code.

You can implement interface methods with static methods (as in the code above) or with virtual methods. You can override virtual methods in subclasses by using the `override` directive. If you don't use virtual methods, you can still provide a new implementation in a subclass by redeclaring the interface type in the subclass, rebinding the interface methods to new versions of the static methods. At first sight, using virtual methods to implement interfaces seems to allow for smoother coding in subclasses, but both approaches are equally powerful and flexible. However, the use of virtual methods affects code size and memory.

NOTE The compiler has to generate stub routines to fix up the interface call entry points to the matching method of the implementing class, and adjust the `self` pointer. The interface method stubs for static methods are very simple: adjust `self` and jump to the real method in the class. The interface method stubs for virtual methods are much more complicated, requiring about four times more code (20 to 30 bytes) in each stub than the static case. Also, adding more virtual methods to the implementing class just bloats the virtual method table (VMT) that much more in the implementing class and all its descendents. Interfaces already have their own VMT, and redeclaring interfaces in descendents to rebind the interface to new methods in the descendent is just as polymorphic as using virtual methods, but much smaller in code size.

Now that we have defined an implementation of the interface, we can write some code to use an object of this class, as usual:

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  Airplane1.Fly;
  Airplane1.Free;
end;
```

But we can also use an interface-type variable:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

As soon as you assign an object to an interface-type variable, Delphi automatically checks to see whether the object implements that interface, using the `as` operator. You can explicitly express this operation as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

NOTE The compiler generates different code for the `as` operator when used with interfaces or with classes. With classes, the compiler introduces run-time checks to verify that the object is effectively "type-compatible" with the given. With interfaces, the compiler sees at compile time that it can extract the necessary interface from the available class type, so it does. This operation is like a "compile-time `as`," not something that exists at run time.

Whether we use the direct assignment or the `as` statement, Delphi does one extra thing: it calls the `_AddRef` method of the object (defined by `IInterface` and implemented by `TInterfacedObject`), increasing its reference count. At the same time, as soon as the

Flyer1 variable goes out of scope, Delphi calls the _Release method (again part of IInterface), which decreases the reference count, checks whether the reference count is zero, and if necessary, destroys the object. For this reason in the listing above, there is no code to free the object we've created.

In other words, in Delphi, objects referenced by interface variables are reference-counted, and they are automatically de-allocated when no interface variable refers to them any more.

WARNING When using interface-based objects, you should generally access them only with object variables or only with interface variables. Mixing the two approaches breaks the reference counting scheme provided by Delphi and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables.

Interface Properties, Delegation, Redefinitions, Aggregation, and Reference Counting Blues

To demonstrate a few technical elements related to interfaces, I've written the IntfDemo example. This example is based on two different interfaces, IWalker and IJumper, defined as follows:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
property Position: Integer read GetPos write SetPos;
end;

IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
property Position: Integer read GetPos write SetPos;
end;
```

Notice that the first interface also defines a property. An interface property is just a name mapped to a read and a write method. You cannot map an interface property to a field, simply because an interface cannot have a data field.

Here comes a sample implementation of the `IWalker` interface. Notice that you don't have to define the property, only its access methods:

```
TRunner = class (TInterfacedObject, IWalker)
private
  Pos: Integer;
public
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

The code is trivial, so I'm going to skip it (you can find it in the IntfDemo example, where there is also a destructor showing a message, used to verify that reference counting works properly). I've implemented the same interface also in another class, `TAthlete`, that I'll discuss in a second.

As I want to implement also the `IJumper` interface in two different classes, I've followed a different approach. Delphi allows you to delegate the implementation of an interface inside a class to an object exposed with a property. In other words, I want to share the actual implementation code for an interface implemented by several unrelated classes.

To support this technique, Delphi has a special keyword, `implements`. For example, you can write:

```
TMyJumper = class (TInterfacedObject, IJumper)
private
  fJumpImpl: IJumper;
public
  constructor Create;
  property Jumper: IJumper read fJumpImpl implements IJumper;
end;
```

In this case the property refers to an interface variable, but you can also use a plain object variable (my preferred approach). The constructor is required for initializing the internal *implementation* object:

```
constructor TMyJumper.Create;
begin
  fJumpImpl := TJumperImpl.Create;
end;
```

As a first attempt (and in the last edition of the book), I defined the implementation class as follows:

```
TJumperImpl = class (TInterfacedObject, IJumper)
private
  Pos: Integer;
public
  function Jump: string;
```

```
    function Walk: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
  end;
```

If you try this code, the program will compile and everything will run smoothly, until you try to check out what happens with reference counting. It won't work, period. The problem lies in the fact that when the program extracts the `IJumper` interface from the `TMyJumper` object, it actually increases and decreases the reference counting of the inner object, instead of the external one. In other words, you have a single compound object and two separate reference counts going on. This can lead to objects being both kept in memory and released too soon.

The solution to this problem is to have a single reference count, by redirecting the `_AddRef` and `_Release` calls of the internal object to the external one (actually we need to do the same also for `QueryInterface`). In the example, I've used the `TAggregatedObject` provided in Delphi 6 by the system unit; refer to the sidebar "Implementing Aggregates" for more details.

As a result of this approach, the implementation class is now defined as follows:

```
    TJumperImpl = class (TAggregatedObject, IJumper)
    private
      Pos: Integer;
    public
      function Jump: string;
      function Walk: string;
      procedure SetPos (Value: Integer);
      function GetPos: Integer;

      property Position: Integer read GetPos write SetPos;
    end;
```

An object using this class for implementing the `IJumper` interface must have a `Create` constructor, to create the internal object, and a destructor, to destroy it. The constructor of the aggregate object requires the container object as parameter, so that it can redirect back the `IInterface` calls. The key element, of course, is the property mapped to the interface with the `implements` keyword:

```
    TMyJumper = class (TInterfacedObject, IJumper)
    private
      fJumpImpl: TJumperImpl;
    public
      constructor Create;
      property Jumper: TJumperImpl read fJumpImpl implements IJumper;
      destructor Destroy; override;
    end;

  constructor TMyJumper.Create;
  begin
    fJumpImpl := TJumperImpl.Create (self);
  end;
```

This example is simple, but in general, things get more complex as you start to modify some of the methods or add other methods that still operate on the data of the internal fJumpImpl object. This final step is demonstrated, along with other features, by the TAthlete class, which implements both the IWalker and IJumper interfaces:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)
private
  fJumpImpl: TJumperImpl;
public
  constructor Create;
  destructor Destroy; override;
  function Run: string; virtual;
  function Walk1: string; virtual;
  function IWalker.Walk = Walk1;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Jumper: TJumperImpl read fJumpImpl implements IJumper;
end;
```

One of the interfaces is implemented directly, whereas the other is delegated to the internal fJumpImpl object. Notice also that by implementing two interfaces that have a method in common, we end up with a name clash. The solution is to rename one of the methods, with the statement

```
function IWalker.Walk = Walk1;
```

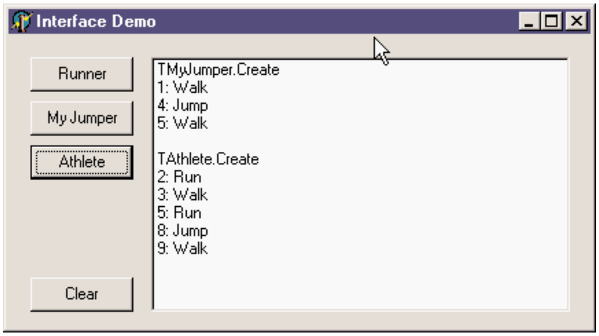
This declaration indicates that the class implements the Walk method of the IWalker interface with a method called Walk1 (instead of with a method having the same name). Finally, in the implementation of all of the methods of this class, we need to refer to the Position property of the fJumpImpl internal object. By declaring a new implementation for the Position property, we'll end up with two positions for a single athlete, a rather odd situation. Here are a couple of examples:

```
function TAthlete.GetPos: Integer;
begin
  Result := fJumpImpl.Position;
end;

function TAthlete.Run: string;
begin
  fJumpImpl.Position := fJumpImpl.Position + 2;
  Result := IntToStr (fJumpImpl.Position) + ': Run';
end;
```

You can further experiment with the IntfDemo example, which has a simple form with buttons to create and call methods of the various objects. Nothing fancy, though, as you can see in Figure 3.4. Simply keep in mind that each call returns the position after the requested movement and a description of the movement itself. Also, each object notifies with a message when it is destroyed.

FIGURE 3.4:

The IntfDemo example



```
TMyJumper.Create
1: Walk
4: Jump
5: Walk

TAthlete.Create
2: Run
3: Walk
5: Run
8: Jump
9: Walk
```

Implementing Aggregates

As mentioned, when you want to use an internal object to implement an interface, you are faced with reference counting problems. Of course, you can provide your own version of the _AddRef and _Release methods of IInterface, but having a ready-to-use solution might help. In fact, QueryInterface on the internal object must also be reflected to the outer object. The user of the interface (whether it works on the outer object or the internal one) should never be able to discern any difference in behavior between _AddRef, _Release, and QueryInterface calls on the aggregated interface and any other interface obtained from the implementing class.

Borland provides a solution to this problem with the TAggregatedObject class. In past version of Delphi, this was defined in the ComObj unit, but now it has been moved into the System unit, to make this feature also available to Linux and to separate it completely from COM support.

The TAggregatedObject class keeps a reference to the controller, the external object, passed as parameter in the constructor. This *weak reference* is kept using a pointer type variable to avoid artificially increasing the reference count of the controller from the aggregated object, something that will prevent the object's reference count from reaching zero. You create an object of this type (used as internal object) passing the reference to the controller (the external object), and all of the IInterface methods are passed back to the controller. A similar class, TContainedObject, lets the controller resolve reference counting, but handles the QueryInterface call internally, limiting the type resolution only to interfaces supported by the internal object.

Working with Exceptions

Another key feature of Object Pascal I'll cover in this chapter is the support for *exceptions*. The idea of exceptions is to make programs more robust by adding the capability of handling software or hardware errors in a uniform way. A program can survive such errors or terminate gracefully, allowing the user to save data before exiting. Exceptions allow you to separate the error-handling code from your normal code, instead of intertwining the two. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

Another benefit is that exceptions define a uniform and universal error-reporting mechanism, which is also used by Delphi components. At run time, Delphi raises exceptions when something goes wrong (in the run-time code, in a component, in the operating system). From the point of the code in which it is raised, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, Delphi handles it, by displaying a standard error message and trying to continue the program, by handing the next system message or user request.

The whole mechanism is based on four keywords:

try delimits the beginning of a protected block of code.

except delimits the end of a protected block of code and introduces the exception-handling statements, with this syntax form:

```
on exception-type do statement
```

finally is used to specify blocks of code that must always be executed, even when exceptions occur. This block is generally used to perform cleanup operations that should always be executed, such as closing files or database tables, freeing objects, and releasing memory and other resources acquired in the same program block.

raise is the statement used to generate an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also `raise` exceptions in your own code when it discovers invalid or inconsistent data at run time. The `raise` keyword can also be used inside a handler to *re-raise* an exception; that is, to propagate it to the next handler.

The most important element to notice up front is that exception handling is no substitute for `if` statements or for tests on input parameters of functions. So *in theory* we could write this code:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
      Result := 0;
  end;
end;
```

In practice, however, this is certainly not a good way of writing your programs. The except block above, like most of the except blocks of the simple examples presented here, has almost no sense at all. In the code above, you should probably not handle the exception but let the program display the error message to the user. An algorithm calling this DivideTwicePlusOne function should not continue (with a meaningless zero value) when this internal error is encountered.

Program Flow and the *finally* Block

But how do we stop the algorithm? The power of exceptions in Delphi relates to the fact that they are "passed" from a routine or method to the calling one, up to a global handler (if the program provides one, as Delphi applications generally do). So the real problem you might have is not how to stop an exception but how to execute some code when an exception is raised.

Consider this method (part of the TryFinally example from the CD), which performs some time-consuming operations and uses the hourglass cursor to show the user that it's doing something:

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;
  MessageDlg ('Total: ' + IntToStr (J), mtInformation, [mbOK], 0);
  Screen.Cursor := crDefault;
end;
```

Because there is an error in the algorithm (as the variable I can reach a value of 0 and is also used in a division), the program will break, but it won't reset the default cursor. This is what a try/finally block is for:

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
```

```
    Screen.Cursor := crHourglass;
    J := 0;
    try
      // long (and wrong) computation...
      for I := 1000 downto 0 do
        J := J + J div I;
      MessageDlg ('Total: ' + IntToStr (J), mtInformation, [mbOK], 0);
    finally
      Screen.Cursor := crDefault;
    end;
  end;
```

When the program executes this function, it always resets the cursor, whether an exception (of any sort) occurs or not.

This code doesn't handle the exception; it merely makes the program robust in case an exception is raised. As a `try` block can be followed by either an `except` or a `finally` statement, but not both of them at the same time, the typical solution if you want to also handle the exception is to use two nested `try` blocks. In this case, you associate the internal one with a `finally` statement and the external one with an `except` statement, or vice versa as the situation requires. Here is the code of this third button of the TryFinally example:

```
  procedure TForm1.BtnTryTryClick(Sender: TObject);
  var
    I, J: Integer;
  begin
    Screen.Cursor := crHourglass;
    J := 0;
    try try
      // long (and wrong) computation...
      for I := 1000 downto 0 do
        J := J + J div I;
      MessageDlg ('Total: ' + IntToStr (J), mtInformation, [mbOK], 0);
    finally
      Screen.Cursor := crDefault;
    end;
    except
      on E: EDivByZero do
      begin
        // re-raise the exception with a new message
        raise Exception.Create ('Error in Algorithm');
      end;
    end;
  end;
```

Every time you have some finalization code at the end of a method, you should place this code in a `finally` block. You should always, invariably, and continuously (how can I stress this more?) protect your code with `finally` statements, to avoid resource or memory leaks in case an exception is raised.

TIP

Handling the exception is generally much less important than using finally blocks, since Delphi can survive most of them. And too many exception-handling blocks in your code probably indicate errors in the program flow and possibly a misunderstanding of the role of exceptions in the language. In the examples in the rest of the book you'll see many try/finally blocks, a few raise statements, and almost no try/except blocks.

Exception Classes

In exception-handling statements shown earlier, we caught the EDivByZero exception, which is defined by Delphi's RTL. Other such exceptions refer to run-time problems (such as a wrong dynamic cast), Windows resource problems (such as out-of-memory errors), or component errors (such as a wrong index). Programmers can also define their own exceptions; you can create a new subclass of the default exception class or one of its subclasses:

```
type
  EArrayFull = class (Exception);
```

When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

This Create method (inherited from the Exception class) has a string parameter to describe the exception to the user. You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

The code presented in the previous excerpts is part of a sample program, called Exception1. Some of the routines have actually been slightly modified, as in the following DivideTwicePlusOne function:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero corrected.', mtError, [mbOK], 0);
    end;
    on E: Exception do
    begin
      Result := 0;
```

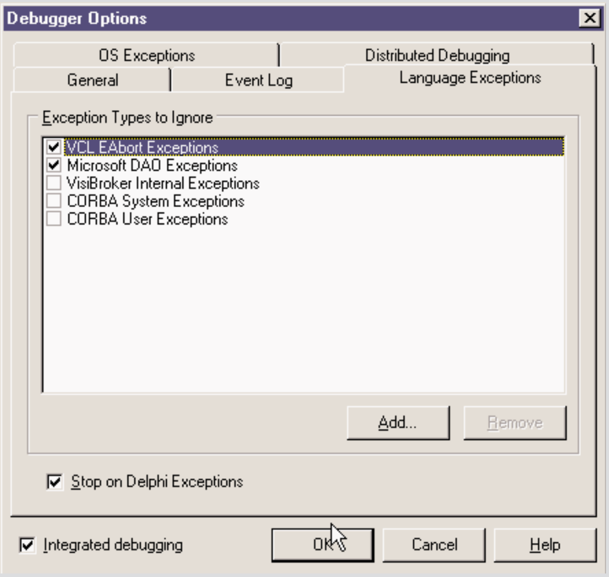
```
   MessageDlg (E.Message, mtError, [mbOK], 0);
  end;
 end; // end except
end;
```

Debugging and Exceptions

When you start a program from the Delphi environment (for example, by pressing the F9 key), you'll generally run it within the debugger. When an exception is encountered, the debugger will stop the program by default. This is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step-by-step. You can also use the Stack Trace feature of Delphi to see the sequence of function and method calls, which caused the program to raise an exception.

In the case of the Exception1 test program, however, this behavior will confuse the program's execution. In fact, even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step-by-step through the code, you can see how it is handled.

If you just want to let the program run when the exception is properly handled, run the program from Windows Explorer, or temporarily disable the Stop on Delphi Exceptions options in the Language Exceptions page of the Debugger Options dialog box (activated by the Tools ➢ Debugger Options command), shown in the Language Exceptions page of the Debugger Options dialog box shown here.



Exception Types to Ignore

☑ VCL EAbort Exceptions
☑ Microsoft DAO Exceptions
☐ VisiBroker Internal Exceptions
☐ CORBA System Exceptions
☐ CORBA User Exceptions

☑ Stop on Delphi Exceptions

☑ Integrated debugging

In the Exception1 code there are two different exception handlers after the same try block. You can have any number of these handlers, which are evaluated in sequence. For this reason, you need to place the broader handlers (the handlers of the ancestor Exception classes) at the end.

In fact, using a hierarchy of exceptions, a handler is also called for the subclasses of the type it refers to, as any procedure will do. This is polymorphism in action again. But keep in mind that using a handler for every exception, such as the one above, is not usually a good choice. It is better to leave unknown exceptions to Delphi. The default exception handler in the VCL displays the error message of the exception class in a message box, and then resumes normal operation of the program. You can actually modify the normal exception handler with the Application.OnException event, as demonstrated in the ErrorLog example later in this chapter.

Another important element of the code above is the use of the exception object in the handler (see on E: Exception do). The object E of class Exception receives the value of the exception object passed by the raise statement. When you work with exceptions, remember this rule: You raise an exception by creating an object and handle it by indicating its type. This has an important benefit, because as we have seen, when you handle a type of exception, you are really handling exceptions of the type you specify as well as any descendant type.

Delphi defines a hierarchy of exceptions, and you can choose to handle each specific type of exception in a different way or handle groups of them together.

Logging Errors

Most of the time, you don't know which operation is going to raise an exception, and you cannot (and should not) wrap each and every piece of code in a try/except block. The general approach is to let Delphi handle all the exceptions and eventually pass them all to you, by handling the OnException event of the global Application object. This can be done rather easily with the ApplicationEvents component.

In the ErrorLog example, I've added to the main form a copy of the ApplicationEvents component and added a handler for its OnException event:

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);
var
  Filename: string;
  LogFile: TextFile;
begin
  // prepares log file
  Filename := ChangeFileExt (Application.Exename, '.log');
  AssignFile (LogFile, Filename);
  if FileExists (FileName) then
    Append (LogFile) // open existing file
  else
    Rewrite (LogFile); // create a new one
  // write to the file and show error
```

```
    Writeln (LogFile, DateTimeToStr (Now) + ':' + E.Message);
    if not CheckBoxSilent.Checked then
      Application.ShowException (E);
    // close the file
    CloseFile (LogFile);
  end;
```

NOTE The ErrorLog example uses the text file support provided by the traditional Turbo Pascal TextFile data type. You can assign a text file variable to an actual file and then read or write it. You can find more on TextFile operations in Chapter 12 of *Essential Pascal*, available on the companion CD.

In the global exceptions handler, you can write to the log, for example, the date and time of the event, and also decide whether to show the exception as Delphi usually does (executing the `ShowException` method of the `TApplication` class). In fact, Delphi by default executes `ShowException` only if there is no `OnException` handler installed.

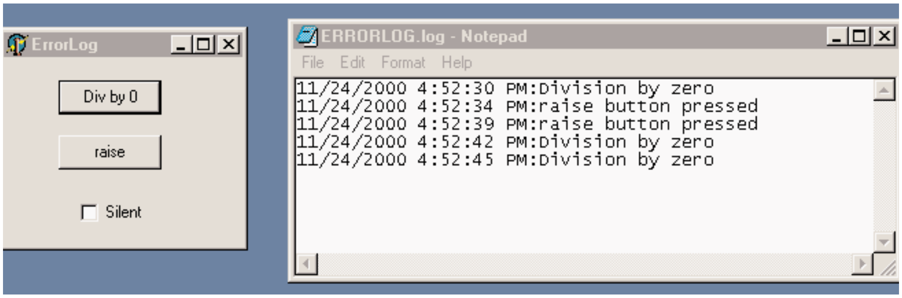
Finally, remember to close the file, flushing the buffers, every time the exception is handled or when the program terminates. I've chosen the first approach to avoid keeping the log file open for the lifetime of the application, potentially making it difficult to work on it. You can accomplish this in the `OnDestroy` event handler of the form:

```
procedure TFormLog.FormDestroy(Sender: TObject);
begin
  CloseFile (LogFile);
end;
```

The form of the program includes a check box to determine its behavior and two buttons generating exceptions. In Figure 3.5, you can see the ErrorLog program running and a sample exceptions log open in Notepad.

FIGURE 3.5:

The ErrorLog example and the log it produces



```
11/24/2000 4:52:30 PM:Division by zero
11/24/2000 4:52:34 PM:raise button pressed
11/24/2000 4:52:39 PM:raise button pressed
11/24/2000 4:52:42 PM:Division by zero
11/24/2000 4:52:45 PM:Division by zero
```

Class References

The final language feature I want to discuss in this chapter is *class references*, which implies the idea of manipulating classes themselves (not just class instances) within your code. The first point to keep in mind is that a class reference isn't a class, it isn't an object, and it isn't a reference to an object; it is simply a reference to a class type.

A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code might make this a little clearer. Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. Actually, there are not many such expressions, but the few cases are interesting. The simplest case is the creation of an object. We can rewrite the two lines above as follows:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

This time I've applied the `Create` constructor to the class reference instead of to an actual class; I've used a class reference to create an object of that class.

NOTE Class references remind us of the concept of *metaclass* available in other OOP languages. In Object Pascal, however, a class reference is not itself a class but only a type pointer. Therefore, the analogy with metaclasses (classes describing other classes) is a little misleading. Actually, `TMetaclass` is also the term used in Borland C++Builder.

Class reference types wouldn't be as useful if they didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef` above, you can then assign to it that specific class and any subclass. So if `MyNewClass` is a subclass of my class, you can also write

```
AClassRef := MyNewClass;
```

Delphi declares a lot of class references in the run-time library and the VCL, including the following:

```
TClass = class of TObject;
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

In particular, the `TClass` class reference type can be used to store a reference to any class you write in Delphi, because every class is ultimately derived from `TObject`. The `TFormClass` reference, instead, is used in the source code of most Delphi projects. The `CreateForm` method of the `Application` object, in fact, requires as parameter the class of the form to create:

```
Application.CreateForm(TForm1, Form1);
```

The first parameter is a class reference; the second is a variable that stores a reference to the created object instance.

Finally, when you have a class reference you can apply to it the class methods of the related class. Considering that each class inherits from `TObject`, you can apply to each class reference some of the methods of `TObject`, as we'll see in the next chapter.

Creating Components Using Class References

What is the *practical* use of class references in Delphi? Being able to manipulate a data type at run time is a fundamental element of the Delphi environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what Delphi does for you behind the scenes.) In other words, class references give you polymorphism for object construction.

To give you a better idea of how class references work, I've built an example named ClassRef. The form displayed by this example is quite simple. It has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, push buttons, and edit boxes.

To make this program run properly, you need to change the names of the three components. The form must also have a class reference field:

```
private
  ClassRef: TControlClass;
  Counter: Integer;
```

The first field stores a new data type every time the user clicks one of the three radio buttons. Here is one of the three methods:

```
procedure TForm1.RadioButtonRadioClick(Sender: TObject);
begin
  ClassRef := TRadioButton;
end;
```

The other two radio buttons have OnClick event handlers similar to this one, assigning the value TEdit or TButton to the ClassRef field. A similar assignment is also present in the handler of the OnCreate event of the form, used as an initialization method.

The interesting part of the code is executed when the user clicks the form. Again, I've chosen the OnMouseDown event of the form to hold the position of the mouse click:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  NewCtrl: TControl;
  MyName: String;
begin
  // create the control
  NewCtrl := ClassRef.Create (Self);
  // hide it temporarily, to avoid flickering
  NewCtrl.Visible := False;
  // set parent and position
  NewCtrl.Parent := Self;
  NewCtrl.Left := X;
  NewCtrl.Top := Y;
  // compute the unique name (and caption)
  Inc (Counter);
  MyName := ClassRef.ClassName + IntToStr (Counter);
  Delete (MyName, 1, 1);
  NewCtrl.Name := MyName;
  // now show it
  NewCtrl.Visible := True;
end;
```

The first line of the code for this method is the key. It creates a new object of the class data type stored in the ClassRef field. We accomplish this simply by applying the Create constructor to the class reference. Now you can set the value of the Parent property, set the position of the new component, give it a name (which is automatically used also as Caption or Text), and make it visible.

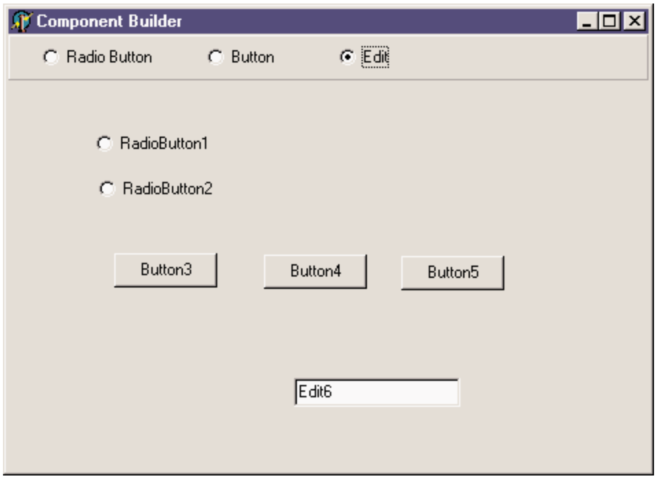
Notice in particular the code used to build the name; to mimic Delphi's default naming convention, I've taken the name of the class with the expression ClassRef.ClassName, using a class method of the TObject class. Then I've added a number at the end of the name and removed the initial letter of the string. For the first radio button, the basic string is TRadioButton, plus the *1* at the end, and minus the *T* at the beginning of the class name—RadioButton1. Sound familiar?

You can see an example of the output of this program in Figure 3.6. Notice that the naming is not exactly the same as used by Delphi. Delphi uses a separate counter for each type of control; I've used a single counter for all of the components. If you place a radio button, a push button, and an edit box in a form of the ClassRef example, their names will be RadioButton1, Button2, and Edit3.

NOTE For polymorphic construction to work, the base class type of the class reference must have a virtual constructor. If you use a virtual constructor (as in the example), the constructor call applied to the class reference will call the constructor of the type that the class reference variable *currently refers to*. But without a virtual constructor, your code will call the constructor of *fixed class type* indicated in the class reference declaration. Virtual constructors are required for polymorphic construction in the same way that virtual methods are required for polymorphism.

FIGURE 3.6:

An example of the output of the ClassRef example



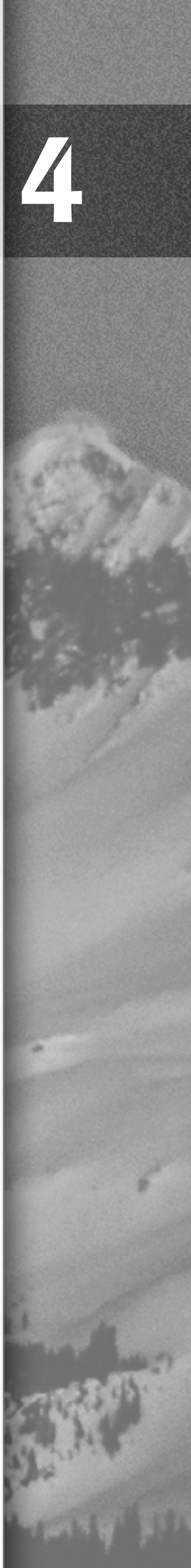
What's Next?

In this chapter, we have discussed the more advanced elements of object-oriented programming in Object Pascal. We have considered inheritance, virtual and abstract methods, polymorphism, safe typecasting, interfaces, exceptions, and class references.

Understanding the secrets of Object Pascal and the structure of the Delphi library is vital for becoming an expert Delphi programmer. These topics form the foundation of working with the VCL and CLX class libraries; after exploring them in the next two chapters, we'll *finally* go on in Part II of the book to explore the development of real applications using all the various components provided by Delphi.

In the meantime, the next chapter will give you an over view of the Delphi run-time library, mainly a collection of functions with little OOP involved. The RTL is an assorted collection of routines and tasks for performing basic tasks with Delphi, and it has been largely extended in Delphi 6.

Chapter 5 will give you more information about the Object Pascal language, discussing features related to the structure of the Delphi class library, such as the effect of the `published` keyword and the role of events. The chapter, as a whole, will discuss the overall architecture of the component library.



CHAPTER 4

The Run-Time Library

- Overview of the RTL
- New Delphi 6 RTL functions
- The conversion engine
- Dates, strings, and other new RTL units
- The *TObject* class
- Showing class information at run time

Delphi uses Object Pascal as its programming language and favors an object-oriented approach, tied with a visual development style. This is where Delphi shines, and we will cover component-based and visual development in this book; however, I want to underline the fact that a lot of ready-to-use features of Delphi come from its run-time library, or RTL for short. This is a large collection of functions you can use to perform simple tasks, as well as some complex ones, within your Pascal code. (I use "Pascal" here, because the run-time library mainly contains procedures and functions and not classes and objects.)

There is actually a second reason to devote this chapter of the book to the run-time library: Delphi 6 sees a large number of enhancements to this area. There are new groups of functions, functions have been moved to new units, and other elements have changed, creating a few incompatibilities with existing code. So even if you've used past versions of Delphi and feel confident with the RTL, you should still read at least portions of this chapter.

The Units of the RTL

As I mentioned above, in Delphi 6 the RTL (run-time library) has a new structure and several new units. The reason for adding new units is that many new functions were added. In most cases, you'll find the existing functions in the units where they used to be, but the new functions will appear in specific units. For example, new functions related to dates are now in the DateUtils unit, but existing date functions have not been moved away from SysUtils in order to avoid incompatibilities with existing code.

The exception to this rule relates to some of the variant support functions, which were moved out of the System unit to avoid unwanted linkage of specific Windows libraries, even in programs that didn't use those features. These variant functions are now part of the new Variants unit, described later in the chapter.

WARNING Some of your existing Delphi code might need to use this new Variants unit to recompile. Delphi 6 is smart enough to acknowledge this and auto-include the Variants unit in projects that use the `Variant` type, issuing only a warning.

A little bit of fine-tuning has also been applied to reduce the minimum size of an executable file, at times enlarged by the unwanted inclusion of global variables or initialization code.

Executable Size under the Microscope

While touching up the RTL, Borland engineers have been able to trim a little "fat" out of each and every Delphi application. Reducing the minimum program size of a few KB seems quite odd, with all the bloated applications you find around these days, but it is a good service to developers. There are cases in which even few KB (multiplied by many applications) can reduce size and eventually download time.

Continued on next page

As a simple test, I've built the MiniSize program, which is not an attempt to build the smallest possible program, but rather an attempt to build a very small program that does something interesting: It reports the size of its own executable file. All of the code of this example is in the source code on the companion CD:

```
program MiniSize;

uses
  Windows;

{$R *.RES}

var
  nSize: Integer;
  hFile: THandle;
  strSize: String;

begin
  // open the current file and read the size
  hFile := CreateFile (PChar (ParamStr (0)),
    0, FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
  nSize := GetFileSize (hFile, nil);
  CloseHandle (hFile);

  // copy the size to a string and show it
  SetLength (strSize, 20);
  Str (nSize, strSize);
  MessageBox (0, PChar (strSize),
    'Mini Program', MB_OK);
end.
```

The program opens its own executable file, after retrieving its name from the first command-line parameter (`ParamStr (0)`), extracts the size, converts it into a string using the simple `Str` function, and shows the result in a message. The program does not have top-level windows. Moreover, I use the `Str` function for the integer-to-string conversion to avoid including SysUtils, which defines all the more complex formatting routines and would impose a little extra overhead.

If you compile this program with Delphi 5, you obtain an executable size of 18,432 bytes. Delphi 6 reduces this size to only 15,360 bytes, trimming about 3 KB. Replacing the long string with a short string, and modifying the code a little, you can trim down the program further, up to 9,216 bytes. This is because you'll end up removing the string support routines and also the memory allocator, something possible only in programs using exclusively low-level calls. You can find both versions in the source code of the example.

Continued on next page

Notice, anyway, that decisions of this type always imply a few trade-offs. In eliminating the overhead of variants from Delphi applications that don't use them, for example, Borland added a little extra burden to applications that do. The real advantage of this operation, though, is in the reduced memory footprint of Delphi applications that do not use variants, as a result of not having to bring in several megabytes of the Ole2 system libraries.

What is really important, in my opinion, is the size of full-blown Delphi applications based on run-time packages. A simple test with a do-nothing program, the MiniPack example, shows an executable size of 15,972 bytes.

In the following sections is a list of the RTL units in Delphi 6, including all the units available (with the complete source code) in the `Source\Rtl\Sys` subfolder of the `Delphi` directory and some of those available in the new subfolder `Source\Rtl\Common`. This new directory hosts the source code of units that make up the new RTL package, which comprises both the function-based library and the core classes, discussed in the next chapter.

NOTE The VCL50 package has now been split into the VCL and RTL packages, so that nonvisual applications using run-time packages don't have the overhead of also deploying visual portions of the VCL. Also, this change helps with Linux compatibility, as the new package is shared between the VCL and CLX libraries. Notice also that the package names in Delphi 6 don't have the version number in their name anymore. When they are compiled, though, the BPL does have the version in its file name, as discussed in more detail in Chapter 12.

I'll give a short overview of the role of each unit and an overview of the groups of functions included. I'll also devote more space to the new Delphi 6 units. I won't provide a detailed list of the functions included, because the online help includes similar reference material. However, I've tried to pick a few interesting or little-known functions, and I will discuss them shortly.

The System and SysInit Units

System is the core unit of the RTL and is automatically included in any compilation (considering an automatic and implicit `uses` statement referring to it). Actually, if you try adding the unit to the `uses` statement of a program, you'll get the compile-time error:

```
[Error] Identifier redeclared: System
```

The System unit includes, among other things:

- The `TObject` class, the base class of any class defined in the Object Pascal language, including all the classes of the VCL. (This class is discussed later in this chapter.)

- The IUnknown and IDispatch interfaces as well as the simple implementation class TInterfacedObject. There are also the new IInterface and IInvokable interfaces. IInterface was added to underscore the point that the interface type in Delphi's Object Pascal language definition is in no way dependent on the Windows operating system (and never has been). IInvokable was added to support SOAP-based invocation. (Interfaces and related classes were introduced in the last chapter and will be discussed further in multiple sections of the book.)
- Some variant support code, including the variant type constants, the TVarData record type and the new TVariantManager type, a large number of variant conversion routines, and also variant records and dynamic arrays support. This area sees a lot of changes compared to Delphi 5. The basic information on variants is provided in Chapter 10 of *Essential Pascal* (available on the companion CD), while an introduction to custom variants is available later in this chapter.
- Many base data types, including pointer and array types and the TDateTime type I've already described in the last chapter.
- Memory allocation routines, such as GetMem and FreeMem, and the actual memory manager, defined by the TMemoryManager record and accessed by the GetMemoryManager and SetMemoryManager functions. For information, the GetHeapStatus function returns a THeapStatus data structure. Two new global variables (AllocMemCount and AllocMemSize) hold the number and total size of allocated memory blocks. There is more on memory and the use of these functions in Chapter 10.
- Package and module support code, including the PackageInfo pointer type, the GetPackageInfoTable global function, and the EnumModules procedure (packages internals are discussed in Chapter 12).
- A rather long list of global variables, including the Windows application instance MainInstance; IsLibrary, indicating whether the executable file is a library or a stand-alone program; IsConsole, indicating console applications; IsMultiThread, indicating whether there are secondary threads; and the command-line string CmdLine. (The unit includes also the ParamCount and ParamStr for an easy access to command-line parameters.) Some of these variables are specific to the Windows platform.
- Thread-support code, with the BeginThread and EndThread functions; file support records and file-related routines; wide string and OLE string conversion routines; and many other low-level and system routines (including a number of automatic conversion functions).

The companion unit of System, called SysInit, includes the system initialization code, with functions you'll seldom use directly. This is another unit that is always implicitly included, as it is used by the System unit.

New in System Unit

I've already described some interesting new features of the System unit in the list above, and most of the changes relate to making the core Delphi RTL more cross-platform portable, replacing Windows-specific features with generic implementations. Along this line, there are new names for interface types, totally revised support for variants, new pointer types, dynamic array support, and functions to customize the management of exception objects.

Another addition for compatibility with Linux relates to line breaks in text files. There is a new `DefaultTextLineBreakStyle` variable, which can be set to either tlbsLF or tlbsCRLF, and a new `sLineBreak` string constant, which has the value #13#10 in the Windows version of Delphi and the value #10 in the Linux version. The line break style can also be set on a file-by-file basis with `SetTextLineBreakStyle` function.

Finally, the System unit now includes the `TFileRec` and `TTextRec` structures, which were defined in the SysUtils unit in earlier versions of Delphi.

The SysUtils and SysConst Units

The SysConst unit defines a few constant strings used by the other RTL units for displaying messages. These strings are declared with the `resourcestring` keyword and saved in the program resources. As other resources, they can be translated by means of the Integrated Translation Manager or the External Translation Manager.

The SysUtils unit is a collection of system utility functions of various types. Different from other RTL units, it is in large part an operating system–dependent unit. The SysUtils unit has no specific focus, but it includes a bit of everything, from string management to locale and multibyte-characters support, from the `Exception` class and several other derived exception classes to a plethora of string-formatting constants and routines.

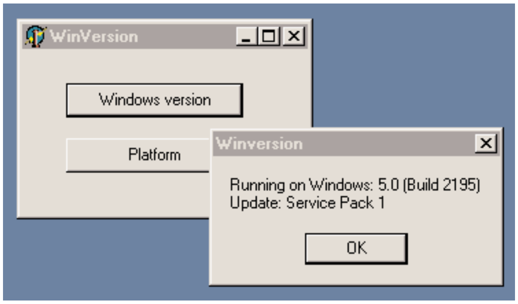
Some of the features of SysUtils are used every day by every programmer as the `IntToStr` or `Format` string-formatting functions; other features are lesser known, as they are the Windows version information global variables. These indicate the Windows platform (Window 9*x* or NT/2000), the operating system version and build number, and the eventual service pack installed on NT. They can be used as in the following code, extracted from the WinVersion example on the companion CD:

```
case Win32Platform of
  VER_PLATFORM_WIN32_WINDOWS:  ShowMessage ('Windows 9x');
  VER_PLATFORM_WIN32_NT:       ShowMessage ('Windows NT');
end;

ShowMessage ('Running on Windows: ' + IntToStr (Win32MajorVersion) + '.' +
  IntToStr (Win32MinorVersion) + ' (Build ' + IntToStr (Win32BuildNumber) +
  ') ' + #10#13 + 'Update: ' + Win32CSDVersion);
```

The second code fragment produces a message like the one in Figure 4.1, depending, of course, on the operating-system version you have installed.

FIGURE 4.1:

The version information displayed by the WinVersion example



WinVersion

Winversion

Running on Windows: 5.0 (Build 2195)
Update: Service Pack 1

Another little-known feature, but one with a rather long name, is a class that supports multithreading: `TMultiReadExclusiveWriteSynchronizer`. This class allows you to work with resources that can be used by multiple threads at the same time for reading (multiread) but must be used by one single thread when writing (exclusive-write). This means that the writing cannot start until all the reading threads have terminated.

NOTE The multi-read synchronizer is unique in that it supports recursive locks and promotion of read locks to write locks. The main purpose of the class is to allow multiple threads easy, fast access to read from a shared resource, but still allow one thread to gain exclusive control of the resource for relatively infrequent updates. There are other synchronization classes in Delphi, declared in the SyncObjs unit and closely mapped to operating-system synchronization objects (such as events and critical sections in Windows).

New SysUtils Functions

Delphi 6 has some new functions within the `SysUtils` unit. One of the new areas relates to Boolean to string conversion. The `BoolToStr` function generally returns '-1' and '0' for true and false values. If the second optional parameter is specified, the function returns the first string in the `TrueBoolStrs` and `FalseBoolStrs` arrays (by default 'TRUE' and 'FALSE'):

```
BoolToStr (True) // returns '-1'
BoolToStr (False, True) // returns 'FALSE' by default
```

The reverse function is `StrToBool`, which can convert a string containing either one of the values of two Boolean arrays mentioned above or a numeric value. In the latter case, the result will be true unless the numeric value is zero. You can see a simple demo of the use of the Boolean conversion functions in the StrDemo example, later in this chapter.

Other new functions of SysUtils relate to floating-point conversions to currency and date time types: `FloatToCurr` and `FloatToDateTime` can be used to avoid an explicit type cast. The `TryStrToFloat` and `TryStrToCurr` functions try to convert a string into a floating point or currency value and will return False in case of error instead of generating an exception (as the classic `StrToFloat` and `StrToCurr` functions do).

There is an `AnsiDequotedStr` function, which removes quotes from a string, matching the `AnsiQuoteStr` function added in Delphi 5. Speaking of strings, Delphi 6 has much-improved support for wide strings, with a series of new routines, including `WideUpperCase`, `WideLowerCase`, `WideCompareStr`, `WideSameStr`, `WideCompareText`, `WideSameText`, and `WideFormat`. All of these functions work like their `AnsiString` counterparts.

Three functions (`TryStrToDate`, `TryEncodeDate`, and `TryEncodeTime`) try to convert a string to a date or to encode a date or time, without raising an exception, similarly to the `Try` functions previously mentioned. In addition, the `DecodeDateFully` function returns more detailed information, such as the day of the week, and the `CurrentYear` function returns the year of today's date.

There is a portable, friendly, overloaded version of the `GetEnvironmentVariable` function. This new version uses string parameters instead of PChar parameters and is definitely easier to use:

```
function GetEnvironmentVariable(Name: string): string;
```

Other new functions relate to interface support. Two new overloaded versions of the little-known `Support` function allow you to check whether an object or a class supports a given interface. The function corresponds to the behavior of the `is` operator for classes and is mapped to the `QueryInterface` method. Here's an example in the code of the IntfDemo program from Chapter 3:

```
var
  W1: IWalker;
  J1: IJumper;
begin
  W1 := TAthlete.Create;
  // more code...
  if Supports (w1, IJumper) then
  begin
    J1 := W1 as IJumper;
    Log (J1.Walk);
  end;
```

There are also an `IsEqualGUID` function and two functions for converting strings to GUIDs and vice versa. The function `CreateGUID` has been moved to SysUtils, as well, to make it also available on Linux (with a custom implementation, of course).

Finally, Delphi 6 has some more Linux-compatibility functions. The AdjustLineBreaks function can now do different types of *adjustments* to carriage-return and line-feed sequences, along with the introduction of new global variables for text files in the System unit, as described earlier. The FileCreate function has an overloaded version in which you can specify file-access rights *the Unix way*. The ExpandFileName function can locate files (on case-sensitive file systems) even when their cases don't exactly correspond. The functions related to path delimiters (backslash or slash) have been made more generic and renamed accordingly. (For example, the IncludeTralingBackslash function is now better known as IncludingTrailingPathDelimiter.)

The Math Unit

The Math unit hosts a collection of mathematical functions: about forty trigonometric functions, logarithmic and exponential functions, rounding functions, polynomial evaluations, almost thirty statistical functions, and a dozen financial functions.

Describing all of the functions of this unit would be rather tedious, although some readers are probably very interested in the mathematical capabilities of Delphi. Here are some of the newer math functions.

New Math Functions

Delphi 6 adds to the Math unit quite a number of new features. There is support for infinite constants (Infinity and NegInfinity) and related comparison functions (IsInfinite and IsNan). There are new trigonometric functions for cosecants and cotangents and new angle-conversion functions.

A handy feature is the availability of an overloaded IfThen function, which returns one of two possible values depending on a Boolean expression. (A similar function is now available also for strings.) You can use it, for example, to compute the minimum of two values:

```
nMin := IfThen (nA < nB, na, nB);
```

NOTE The IfThen function is similar to the ?: operator of the C/C++ language, which I find very handy because you can replace a complete if/then/else statement with a much shorter expression, writing less code and often declaring fewer temporary variables.

The RandomRange and RandomFrom can be used instead of the traditional Random function to have more control on the random values produced by the RTL. The first function returns a number within two extremes you specify, while the second selects a random value from an array of possible numbers you pass to it as a parameter.

The InRange Boolean function can be used to check whether a number is within two other values. The EnsureRange function, instead, forces the value to be within the specified range.

The return value is the number itself or the lower limit or upper limit, in the event the number is out of range. Here is an example:

```
// do something only if value is within min and max
if InRange (value, min, max) then
  ...

// make sure the value is between min and max
value := EnsureRange (value, min, max);
...
```

Another set of very useful functions relates to comparisons. Floating-point numbers are fundamentally inexact; a floating-point number is an approximation of a theoretical real value. When you do mathematical operations on floating-point numbers, the inexactness of the original values accumulates in the results. Multiplying and dividing by the same number might not return exactly the original number but one that is very close to it. The `SameValue` function allows you to check whether two values are close enough in value to be considered equal. You can specify how close the two numbers should be or let Delphi compute a reasonable error range for the representation you are using. (This is why the function is overloaded.) Similarly, the `IsZero` function compares a number to zero, with the same "fuzzy logic."

The `CompareValue` function uses the same rule for floating-point numbers but is available also for integers; it returns one of the three constants `LessThanValue`, `EqualsValue`, and `GreaterThanValue` (corresponding to –1, 0, and 1). Similarly, the new `Sign` function returns –1, 0, and 1 to indicate a negative value, zero, or a positive value.

The `DivMod` function is equivalent to both div and mod operations, returning the result of the integer division and the remainder (or modulus) at once. The `RoundTo` function allows you to specify the rounding digit—allowing, for example, rounding to the nearest thousand or to two decimals:

```
RoundTo (123827, 3);   // result is 124,000
RoundTo (12.3827, -2); // result is 12.38
```

WARNING Notice that the `RoundTo` function uses a positive number to indicate the power of ten to round to (for example, `2` for hundreds) or a negative number for the number of decimal places. This is exactly the opposite of the `Round` function used by spreadsheets such as Excel.

There are also some changes to the standard rounding operations provided by the `Round` function: You can now control how the FPU (the floating-point unit of the CPU) does the rounding by calling the `SetRoundMode` function. There are also functions to control the FPU precision mode and its exceptions.

The New ConvUtils and StdConvs Units

The new ConvUtils unit contains the core of the conversion engine. It uses the conversion constants defined by a second unit, StdConvs. I'll cover these two units later in this chapter, showing you also how to extend them with new measurement units.

The New DateUtils Unit

The DateUtils unit is a new collection of date and time-related functions. It includes new functions for picking values from a TDateTime variable or counting values from a given interval, such as

```
// pick value
function DayOf(const AValue: TDateTime): Word;
function HourOf(const AValue: TDateTime): Word;
// value in range
function WeekOfYear(const AValue: TDateTime): Integer;
function HourOfWeek(const AValue: TDateTime): Integer;
function SecondOfHour(const AValue: TDateTime): Integer;
```

Some of these functions are actually quite odd, such as MilliSecondOfMonth or SecondOfWeek, but Borland developers have decided to provide a complete set of functions, no matter how impractical they sound. I actually used some of these functions in Chapter 2, to build the TDate class.

There are functions for computing the initial or final value of a given time interval (day, week, month, year) including the current date, and for range checking and querying; for example:

```
function DaysBetween(const ANow, AThen: TDateTime): Integer;
function WithinPastDays(const ANow, AThen: TDateTime;
  const ADays: Integer): Boolean;
```

Other functions cover incrementing and decrementing by each of the possible time intervals, encoding and "recoding" (replacing one element of the TDateTime value, such as the day, with a new one), and doing "fuzzy" comparisons (approximate comparisons where a difference of a millisecond will still make two dates equal). Overall, DateUtils is quite interesting and not terribly difficult to use.

The New StrUtils Unit

The StrUtils unit is a new unit with some new string-related functions. One of the key features of this unit is the availability of many new string comparison functions. There are functions based on a "soundex" algorithm (AnsiResembleText), some providing lookup in arrays of strings (AnsiMatchText and AnsiIndexText), sub-string location, and replacement (including AnsiContainsText and AnsiReplaceText).

NOTE *Soundex* is an algorithm to compare names based on how they sound rather then how they are spelled. The algorithm computes a number for each word sound, so that comparing two such numbers you can determine whether two names sound similar. The system was first applied 1880 by the U.S. Bureau of the Census, patented in 1918, and is now in the public domain. The soundex code is an indexing system that translates names into a four-character code consisting of one letter and three numbers. More information is at `www.nara.gov/genealogy/coding.html`.

Beside comparisons, other functions provide a two-way test (the nice `IfThen` function, similar to the one we've already seen for numbers), duplicate and reverse strings, and replace sub-strings. Most of these string functions were added as a convenience to Visual Basic programmers migrating to Delphi.

I've used some of these functions in the StrDemo example on the companion CD, which uses also some of the new Boolean-to-string conversions defined within the SysUtils unit. The program is actually a little more than a test for a few of these functions. For example, it uses the "soundex" comparison between the strings entered in two edit boxes, converting the resulting Boolean into a string and showing it:

```
ShowMessage (BoolToStr (AnsiResemblesText
  (EditResemble1.Text, EditResemble2.Text), True));
```

The program also showcases the `AnsiMatchText` and `AnsiIndexText` functions, after filling a dynamic array of strings (called `strArray`) with the values of the strings inside a list box. I could have used the simpler `IndexOf` method of the `TStrings` class, but this would have defeated the purpose of the example. The two list comparisons are done as follows:

```
procedure TForm1.ButtonMatchesClick(Sender: TObject);
begin
  ShowMessage (BoolToStr (AnsiMatchText(EditMatch.Text, strArray), True));
end;

procedure TForm1.ButtonIndexClick(Sender: TObject);
var
  nMatch: Integer;
begin
  nMatch := AnsiIndexText(EditMatch.Text, strArray);
  ShowMessage (IfThen (nMatch >= 0, 'Matches the string number ' +
    IntToStr (nMatch), 'No match'));
end;
```

Notice the use of the `IfThen` function in the last few lines of code, with two alternative output strings, depending on the result of the initial test (`nMatch <= 0`).

Three more buttons do simple calls to three other new functions, with the following lines of code (one for each):

```
// duplicate (3 times) a string
ShowMessage (DupeString (EditSample.Text, 3));
// reverse the string
ShowMessage (ReverseString (EditSample.Text));
// choose a random string
ShowMessage (RandomFrom (strArray));
```

The New Types Unit

The Types unit is a new Pascal file holding data types common to multiple operating systems. In past versions of Delphi, the same types were defined by the Windows unit; now they've been moved to this common unit, shared by Delphi and Kylix. The types defined here are simple ones and include, among others, the `TPoint`, `TRect`, and `TSmallPoint` record structures plus their related pointer types.

The New Variants and VarUtils Units

Variants and VarUtils are two new variant-related units. The Variants unit contains generic code for variants. As mentioned earlier, some of the routines in this unit have been moved here from the System unit. Functions include generic variant support, variant arrays, variant copying, and dynamic array to variant array conversions. There is also the `TCustomVariantType` class, which defines customizable variant data types.

The Variants unit is totally platform independent and uses the VarUtils unit, which contains OS-dependent code. In Delphi, this unit uses the system APIs to manipulate variant data, while in Kylix it uses some custom code provided by the RTL library.

Custom Variants and Complex Numbers

The possibility to extend the type system with custom variants is brand new in Delphi 6. It allows you to define a new data type that, contrary to a class, overloads standard arithmetic operators.

In fact, a variant is a type holding both type specification and the actual value. A variant can contain a string, another can contain a number. The system defines automatic conversions among variant types, allowing you to mix them inside operations (including custom variants). This flexibility comes at a high cost: operations on variants are much slower than on native types, and variants use extra memory.

As an example of a custom variant type, Delphi 6 ships with an interesting definition for complex numbers, found in the VarCmplx unit (available in source-code format in the

Rtl\Common folder). You can create complex variants by using one of the overloaded VarComplexCreate functions and use them in any expression, as the following code fragment demonstrates:

```
var
  v1, v2: Variant;
begin
  v1 := VarComplexCreate (10, 12);
  v2 := VarComplexCreate (10, 1);
  ShowMessage (v1 + v2 + 5);
```

The complex numbers are actually defined using classes, but they are surfaced as variants by inheriting a new class from the TCustomVariantType class (defined in the Variants unit), overriding a few virtual abstract functions, and creating a global object that takes care of the registration within the system.

Beside these internal definitions, the unit includes a long list of routines for operating on variant, including mathematical and trigonometric operations. I'll leave them to your study, as not all readers may be interested in complex numbers for their programs.

WARNING Building a custom variant is certainly not an easy task, and I can hardly find reasons for using them instead of objects and classes. In fact, with a custom variant you gain the advantage of using operator overloading on your own data structures, but you lose compile-time checking, make the code much slower, miss several OOP features, and have to write a lot of rather complex code.

The DelphiMM and ShareMem Units

The DelphiMM and ShareMem units relate to memory management. The actual Delphi memory manager is declared in the System unit. The DelphiMM unit defines an alternative memory manager library to be used when passing strings from an executable to a DLL (a Windows dynamic linking library), both built with Delphi.

The interface to this memory manager is defined in the ShareMem unit. This is the unit you must include (compulsory as first unit) in the projects of both your executable and library (or libraries). Then, you'll also need to distribute and install the Borlndmm.dll library file along with your program.

COM-Related Units

ComConts, ComObj, and ComServ provide low-level COM support. As these units are not really part of the RTL, from my point of view, I won't discuss them here in any detail. You can refer to Chapter 20 for all the related information. In any case, these units have not changed a lot since the last version of Delphi.

Converting Data

Delphi 6 includes a new conversion engine, defined in the ConvUtils unit. The engine by itself doesn't include any definition of actual measurement units; instead, it has a series of core functions for end users.

The key function is the actual conversion call, the `Convert` function. You simply provide the amount, the units it is expressed in, and the units you want it converted into. The following would convert a temperature of 31 degrees Celsius to Fahrenheit:

```
Convert (31, tuCelsius, tuFahrenheit)
```

An overloaded version of the `Convert` function allows converting values that have two units, such as speed (which has both a length and a time unit). For example, you can convert miles per hours to meters per second with this call:

```
Convert (20, duMiles, tuHours, duMeters, tuSeconds)
```

Other functions in the unit allow you to convert the result of an addition or a difference, check if conversions are applicable, and even list the available conversion families and units.

A predefined set of measurement units is provided in the StdConvs unit. This unit has conversion families and an impressive number of actual values, as in the following reduced excerpt:

```
  // Distance Conversion Units
  // basic unit of measurement is meters
  cbDistance: TConvFamily;

  duAngstroms: TConvType;
  duMicrons: TConvType;
  duMillimeters: TConvType;
  duMeters: TConvType;
  duKilometers: TConvType;
  duInches: TConvType;
  duMiles: TConvType;
  duLightYears: TConvType;
  duFurlongs: TConvType;
  duHands: TConvType;
  duPicas: TConvType;
```

This family and the various units are registered in the conversion engine in the initialization section of the unit, providing the conversion ratios (saved in a series of constants, as `MetersPerInch` in the code below):

```
cbDistance := RegisterConversionFamily('Distance');
duAngstroms := RegisterConversionType(cbDistance, 'Angstroms', 1E-10);
duMillimeters := RegisterConversionType(cbDistance, 'Millimeters', 0.001);
duInches := RegisterConversionType(cbDistance, 'Inches', MetersPerInch);
```

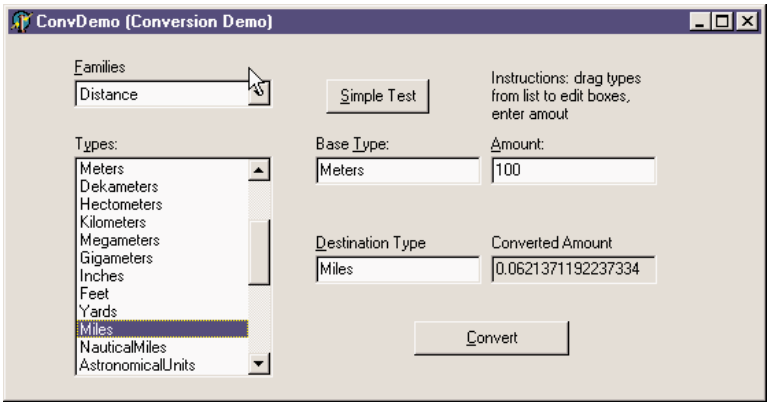
To test the conversion engine, I built a generic example (ConvDemo on the companion CD) that allows you to work with the entire set of available conversions. The program fills a combo box with the available conversion families and a list box with the available units of the active family. This is the code:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  GetConvFamilies (aFamilies);
  for i := Low(aFamilies) to High(aFamilies) do
    ComboFamilies.Items.Add (ConvFamilyToDescription (aFamilies[i]));
  // get the first and fire event
  ComboFamilies.ItemIndex := 0;
  ChangeFamily (self);
end;

procedure TForm1.ChangeFamily(Sender: TObject);
var
  aTypes: TConvTypeArray;
  i: Integer;
begin
  ListTypes.Clear;
  CurrFamily := aFamilies [ComboFamilies.ItemIndex];
  GetConvTypes (CurrFamily, aTypes);
  for i := Low(aTypes) to High(aTypes) do
    ListTypes.Items.Add (ConvTypeToDescription (aTypes[i]));
end;
```

The `aFamilies` and `CurrFamily` variables are declared in the private section of the form as follows:

```
aFamilies: TConvFamilyArray;
CurrFamily: TConvFamily;
```

At this point, a user can enter two measurement units and an amount in the corresponding edit boxes of the form, as you can see in Figure 4.2. To make the operation faster, it is actually possible to select a value in the list and drag it to one of the two Type edit boxes. The dragging support is described in the sidebar "Simple Dragging in Delphi."



ConvDemo (Conversion Demo)

Instructions: drag types from list to edit boxes, enter amout

Families: Distance

Types:

Meters
Dekameters
Hectometers
Kilometers
Megameters
Gigameters
Inches
Feet
Yards
Miles
NauticalMiles
AstronomicalUnits

Base Type: Meters

Amount: 100

Destination Type: Miles

Converted Amount: 0.0621371192237334

FIGURE 4.2: The ConvDemo example at run time

Simple Dragging in Delphi

The ConvDemo example I've built to show how to use the new conversion engine of Delphi 6 uses an interesting technique: dragging. In fact, you can move the mouse over the list box, select an item, and then keep the left mouse button pressed and drag the item over one of the edit boxes in the center of the form.

To accomplish this, I had to set the `DragMode` property of the list box (the source component) to dmAutomatic and implement the `OnDragOver` and `OnDragDrop` events of the target edit boxes (the two edit boxes are connected to the same event handlers, sharing the same code). In the first method, the program indicates that the edit boxes always accept the dragging operation, regardless of the source. In the second method, the program copies the text selected in the list box (the `Source` control of the dragging operation) to the edit box that fired the event (the `Sender` object). Here is the code for the two methods:

```
procedure TForm1.EditTypeDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;
end;

procedure TForm1.EditTypeDragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  (Sender as TEdit).Text := (Source as TListBox).Items
    [(Source as TListBox).ItemIndex];
end;
```

The units must match those available in the current family. In case of error, the text of the Type edit boxes is shown in red. This is the effect of the first part of the DoConvert method of the form, which is activated as soon as the value of one of the edit boxes for the units or the amount changes. After checking the types in the edit boxes, the DoConvert method does the actual conversion, displaying the result in the fourth, grayed edit box. In case of errors, you'll get a proper message in the same box. Here is the code:

```
procedure TForm1.DoConvert(Sender: TObject);
var
  BaseType, DestType: TConvType;
begin
  // get and check base type
  if not DescriptionToConvType(CurrFamily, EditType.Text, BaseType) then
    EditType.Font.Color := clRed
  else
    EditType.Font.Color := clBlack;

  // get and check destination type
  if not DescriptionToConvType(CurrFamily, EditDestination.Text,
      DestType) then
    EditDestination.Font.Color := clRed
  else
    EditDestination.Font.Color := clBlack;

  if (DestType = 0) or (BaseType = 0) then
    EditConverted.Text := 'Invalid type'
  else
    EditConverted.Text := FloatToStr (Convert (
      StrToFloat (EditAmount.Text), BaseType, DestType));
end;
```

If all this is not interesting enough for you, consider that the conversion types provided serve only as a demo: You can fully customize the engine, by providing the measurement units you are interested in, as described in the next section.

What About Currency Conversions?

Converting currencies is not exactly the same as converting measurement units, as currency rates change at very high speed. In theory, you can register a conversion rate with Delphi's conversion engine. From time to time, you check the new rate exchange, unregister the existing conversion, and register a new one. However, keeping up with the actual rate means changing the conversion so often that the operation might not make a lot of sense. Also, you'll have to triangulate conversions: you have to define a base unit (probably the U.S. dollar if you live in America) and convert to and from this currency even for converting between two different ones.

What's more interesting is to use the engine for converting member currencies of the euro, for two reasons. First, conversion rates are fixed (until the single euro currency actually takes over). Second, the conversion among euro currencies is legally done by converting a currency to euros first and then from the euro amount to the other currency, the exact behavior of Delphi's conversion engine. There is only a small problem, as you should apply a rounding algorithm at every step of the conversion. I'll consider this problem after I've provided the base code for integrating euro currencies with Delphi 6 conversion engine.

NOTE The ConvertIt demo of Delphi 6 provides support for euro conversions, using a slightly different rounding approach (which might be more correct or not, I'm not really sure). I've decided to keep this example anyway, as it is instructive in showing how to create a new measurement system (and I lacked another example as good).

The example, called EuroConv, is actually meant to teach how to register any new measurement unit with the engine. Following the template provided by the StdConvs unit, I've created a new unit (called EuroConvConst) and in the interface portion I've declared variables for the family and the specific units, as follows:

```
interface

var
  // Euro Currency Conversion Units
  cbEuroCurrency: TConvFamily;

  cuEUR: TConvType;
  cuDEM: TConvType; // Germany
  cuESP: TConvType; // Spain
  cuFRF: TConvType; // France
  cuIEP: TConvType; // Ireland
  cuITL: TConvType; // Italy
  // and so on...
```

In the implementation portion of the unit, I've defined constants for the various official conversion rates:

```
implementation

const
  DEMPerEuros = 1.95583;
  ESPPerEuros = 166.386;
  FRFPerEuros = 6.55957;
  IEPPerEuros =  0.787564;
  ITLPerEuros =  1936.27;
  // and so on...
```

Finally, in the unit initialization code I've registered the family and the various currencies, each with its own conversion rate and a readable name:

```
initialization
  // Euro Currency's family type
  cbEuroCurrency := RegisterConversionFamily('EuroCurrency');

  cuEUR := RegisterConversionType(
    cbEuroCurrency, 'EUR', 1);
  cuDEM := RegisterConversionType(
    cbEuroCurrency, 'DEM', 1 / DEMPerEuros);
  cuESP := RegisterConversionType(
    cbEuroCurrency, 'ESP', 1 / ESPPerEuros);
  cuFRF := RegisterConversionType(
    cbEuroCurrency, 'FRF', 1 / FRFPerEuros);
  cuIEP := RegisterConversionType(
    cbEuroCurrency, 'IEP', 1 / IEPPerEuros);
  cuITL := RegisterConversionType(
    cbEuroCurrency, 'ITL', 1 / ITLPerEuros);
```

NOTE The engine uses as a conversion factor the amount of the base unit to obtain the secondary ones, with a constant like `MetersPerInch`, for example. The standard rate of euro currencies is defined in the opposite way. For this reason, I've decided to keep the conversion constants with the official values (as `DEMPerEuros above`) and pass them to the engine as fractions (`1/DEMPerEuros`).

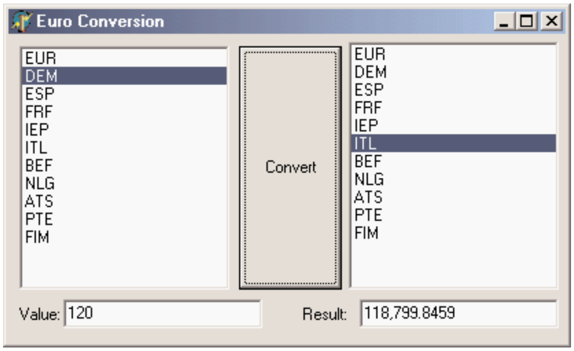
Having registered this unit, we can now convert 120 German marks to Italian liras by writing:

```
Convert (120, cuDEM, cuITL)
```

The demo program actually does a little more, providing two list boxes with the available currencies, extracted as in the previous example, and edit boxes for the input value and final result. You can see the form at run time in Figure 4.3.

FIGURE 4.3:

The output of the EuroConv unit, showing the use of Delphi's conversion engine with a custom measurement unit



Value: 120

Result: 118,799.8459

The program works nicely but is not perfect, as the proper rounding is not applied. In fact, you should round not only the final result of the conversion but also the intermediate value. Using the conversion engine to accomplish this directly is not easy. The engine allows you to provide either a custom conversion function or a conversion rate. But writing identical conversion functions for the all the various currencies seems a bad idea, so I've decided to go a different path. (You can see examples of custom conversion functions in the StdConvs unit, in the portion related to temperatures.)

In the EuroConv example, I've added to the unit with the conversion rates a custom function, called `EuroConv`, that does the proper conversion. Simply calling this function instead of the standard `Convert` function does the trick (and I really see no drawback to this approach, because in programs like this, you'll hardly mix currencies with meters or temperatures). As an alternative, I could inherit a new class from `TConvTypeFactor`, providing a new version of the `FromCommon` and `ToCommon` methods, or I could have called the overloaded versions of the `RegisterConversionType` that accepts these two functions as parameters. None of these techniques, however, would have allowed me to handle special cases, such as the conversion of a currency to itself.

This is the code of the `EuroConv` function, which uses the internal `EuroRound` function for rounding to the number of digits specified in the `Decimals` parameter (which must be between 3 and 6, according with the official rules):

```
type
  TEuroDecimals = 3..6;

function EuroConvert (const AValue: Double;
  const AFrom, ATo: TConvType;
  const Decimals: TEuroDecimals = 3): Double;

  function EuroRound (const AValue: Double): Double;
  begin
    Result := AValue * Power (10, Decimals);
    Result := Round (Result);
    Result := Result / Power (10, Decimals);
  end;

begin
  // check special case: no conversion
  if AFrom = ATo then
    Result := AValue
  else
  begin
    // convert to Euro, then round
    Result := ConvertFrom (AFrom, AValue);
    Result := EuroRound (Result);
```

```
      // convert to currency then round again
      Result := ConvertTo (Result, ATo);
      Result := EuroRound (Result);
    end;
  end;
```

Of course, you might want to extend the example by providing conversion to other non-euro currencies, eventually picking the values automatically from a Web site. I'll leave this as a rather complex exercise.

The *TObject* Class

As mentioned earlier, a key element of the System unit is the definition of the `TObject` class, the *mother of all Delphi classes*. Every class in the system is a subclass of the `TObject` class, either directly (for example, if you indicate no base class) or indirectly. The whole hierarchy of the classes of an Object Pascal program has a single root. This allows you to use the `TObject` data type as a replacement for the data type of any class type in the system.

For example, event handlers of components usually have a `Sender` parameter of type `TObject`. This simply means that the `Sender` object can be of any class, since every class is ultimately derived from `TObject`. The typical drawback of such an approach is that to work on the object, you need to know its data type. In fact, when you have a variable or a parameter of the `TObject` type, you can apply to it only the methods and properties defined by the `TObject` class itself. If this variable or parameter happens to refer to an object of the `TButton` type, for example, you cannot directly access its `Caption` property. The solution to this problem lies in the use of the safe down-casting or run-time type information (RTTI) operators (`is` and `as`) discussed in Chapter 3.

There is another approach. For any object, you can call the methods defined in the `TObject` class itself. For example, the `ClassName` method returns a string with the name of the class. Because it is a class method (see Chapter 2 for details), you can actually apply it both to an object and to a class. Suppose you have defined a `TButton` class and a `Button1` object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

There are occasions when you need to use the name of a class, but it can also be useful to retrieve a class reference to the class itself or to its base class. The class reference, in fact, allows you to operate on the class at run time (as we've seen in the preceding chapter), while the class name is just a string. We can get these class references with the `ClassType` and `ClassParent` methods. The first returns a class reference to the class of the object, the second to its base class. Once you have a class reference, you can apply to it any class methods of TObject—for example, to call the ClassName method.

Another method that might be useful is InstanceSize, which returns the run-time size of an object. Although you might think that the SizeOf global function provides this information, that function actually returns the size of an object reference—a pointer, which is invariably four bytes—instead of the size of the object itself.

In Listing 4.1, you can find the complete definition of the TObject class, extracted from the System unit. Beside the methods I've already mentioned, notice InheritsFrom, which provides a test very similar to the is operator but that can be applied also to classes and class references (while the first argument of is must be an object).

Listing 4.1: The definition of the *TObject* class (in the System RTL unit)

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(
      const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID;out Obj): Boolean;
    class function GetInterfaceEntry(
      const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

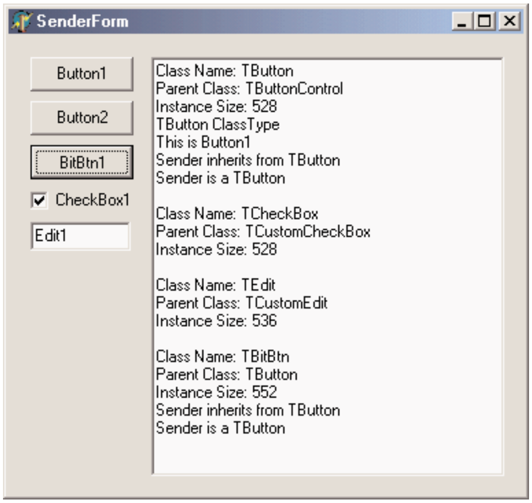
NOTE The ClassInfo method returns a pointer to the internal run-time type information (RTTI) of the class, introduced in the next chapter.

These methods of TObject are available for objects of every class, since TObject is the common ancestor class of every class. Here is how we can use these methods to access class information:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
  Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);

  if Sender.ClassParent <> nil then
    Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);

  Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
end;
```

The code checks to see whether the ClassParent is nil in case you are actually using an instance of the TObject type, which has no base type. This ShowSender method is part of the IfSender example on the companion CD. The method is connected with the OnClick event of several controls: three buttons, a check box, and an edit box. When you click each control, the ShowSender method is invoked with the corresponding control as sender (more on events in the next chapter). One of the buttons is actually a Bitmap button, an object of a TButton subclass. You can see an example of the output of this program at run time in Figure 4.4.

FIGURE 4.4: The output of the IfSender example



```
Class Name: TButton
Parent Class: TButtonControl
Instance Size: 528
TButton ClassType
This is Button1
Sender inherits from TButton
Sender is a TButton

Class Name: TCheckBox
Parent Class: TCustomCheckBox
Instance Size: 528

Class Name: TEdit
Parent Class: TCustomEdit
Instance Size: 536

Class Name: TBitBtn
Parent Class: TButton
Instance Size: 552
Sender inherits from TButton
Sender is a TButton
```

You can use other methods to perform tests. For example, you can check whether the Sender object is of a specific type with the following code:

```
if Sender.ClassType = TButton then ...
```

You can also check whether the Sender parameter corresponds to a given object, with this test:

```
if Sender = Button1 then...
```

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class *or* one of its subclasses. This lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the InheritsFrom method, which is also called when you use the is operator. The following two tests are equivalent:

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

Showing Class Information

I've extended the IfSender example to show a complete list of base classes of a given object or class. Once you have a class reference, in fact, you can add all of its base classes to the ListParent list box with the following code:

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
```

You'll notice that we use a class reference at the heart of the while loop, which tests for the absence of a parent class (so that the current class is TObject). Alternatively, we could have written the while statement in either of the following ways:

```
while not MyClass.ClassNameIs ('TObject') do...
while MyClass <> TObject do...
```

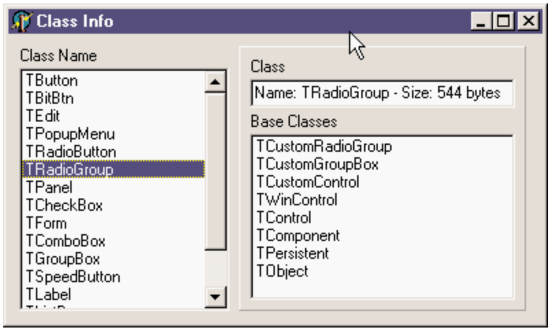
The code in the with statement referring to the ListParent list box is part of the ClassInfo example (see the companion CD), which displays the list of parent classes and some other information about a few components of the VCL, basically those on the Standard page of the Component Palette. These components are manually added to a dynamic array holding classes and declared as

```
private
  ClassArray: array of TClass;
```

When the program starts, the array is used to show all the class names in a list box. Selecting an item from the list box triggers the visual presentation of its details and its base classes, as you can see in the output of the program, in Figure 4.5.

FIGURE 4.5:

The output of the ClassInfo example



```
TButton
TBitBtn
TEdit
TPopupMenu
TRadioButton
TRadioGroup
TPanel
TCheckBox
TForm
TComboBox
TGroupBox
TSpeedButton
TLabel
```

Name: TRadioGroup - Size: 544 bytes

```
TCustomRadioGroup
TCustomGroupBox
TCustomControl
TWinControl
TControl
TComponent
TPersistent
TObject
```

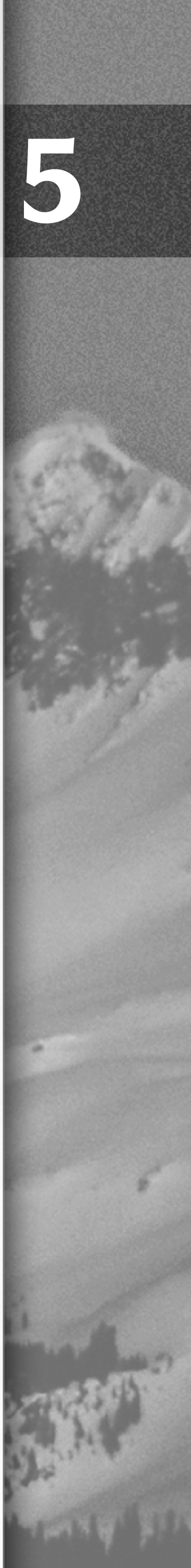
NOTE As a further extension to this example, it is possible to create a tree with all of the base classes of the various components in a hierarchy. To do that, I've created the VclHierarchy program, which you can find on my Web site, `www.marcocantu.com`, in the CanTools section.

What's Next?

In this chapter I've focused my attention on new features of the Delphi 6 function-based run-time library. I have provided only a summary of the entire RTL, not a complete overview, as this would have taken too much space. You can find more examples of the basic RTL functions of Delphi in my free electronic book *Essential Pascal*, which is featured on the companion CD.

In the next chapter, we'll start moving from the function-based RTL to the class-based RTL, which is the core of Delphi's class library. I won't debate whether the core classes common to the VCL and CLX, such as `TObject`, actually belong to the RTL or the class library. I've covered everything defined in System, SysUtils, and other units hosting functions and procedures in this chapter, while the next chapter focuses on the Classes unit and other core units defining classes.

Along with the preceding two chapters on the Object Pascal language, this will provide a foundation for discussing visual- and database-oriented classes, or components, if you prefer. Looking to the various library units, we'll find many more global functions, which don't belong to the core RTL but are still quite useful!



Chapter 5

Core Library Classes

- The RTL package, CLX, and VCL
- *TPersistent* and *published*
- The *TComponent* base class and its properties
- Components and ownership
- Events
- Lists, container classes, and collections
- Streaming
- Summarizing the units of the RTL package

We saw in the preceding chapter that Delphi includes a large number of functions and procedures, but the real power of Delphi's visual programming lies in the huge class library it comes with. Delphi's standard class library contains hundreds of classes, with thousands of methods, and it is so large that I certainly cannot provide a detailed reference in this book. What I'll do, instead, is explore various areas of this library starting with this chapter and continuing through the following ones.

This first chapter is devoted to the core classes of the library as well as to some standard programming techniques, such as the definition of events. We'll explore some commonly used classes, such as lists, string lists, collections, and streams. We'll devote most of our time to exploring the content of the Classes unit, but we'll devote time also to other core units of the library.

Delphi classes can be used either entirely in code or within the visual form designer. Some of them are component classes, which show up in the Component Palette, while others are more general-purpose. The terms *class* and *component* can be used almost as synonyms in Delphi. Components are the central elements of Delphi applications. When you write a program, you basically choose a number of components and define their interactions. That's all there is to Delphi visual programming.

Before reading this chapter, you need to have a good understanding of the Object Pascal programming language, including inheritance, properties, virtual methods, class references, and so on, as discussed in Chapters 2 and 3 of this book.

The RTL Package, VCL, and CLX

Until version 5, Delphi's class library was known as VCL, which stands for Visual Components Library. Kylix, the Delphi version for Linux, introduced a new component library, called CLX (pronounced "clicks" and standing for Component Library for X-Platform or Cross Platform). Delphi 6 includes both the VCL and CLX libraries. For visual components, the two class libraries are alternative one to the other. However, the core classes and the database and Internet portions of the two libraries are basically shared.

VCL was considered as a single large library, although programmers used to refer to different parts of it (components, controls, nonvisual components, data sets, data-aware controls, Internet components, and so on). CLX introduces a distinction in four parts: BaseCLX, VisualCLX, DataCLX, and NetCLX. Only in VisualCLX does the library use a totally different approach between the two platforms, with the rest of the code being inherently portable to Linux. In the following section, I discuss the sections of these two libraries, while the rest of the chapter focuses on the common core classes.

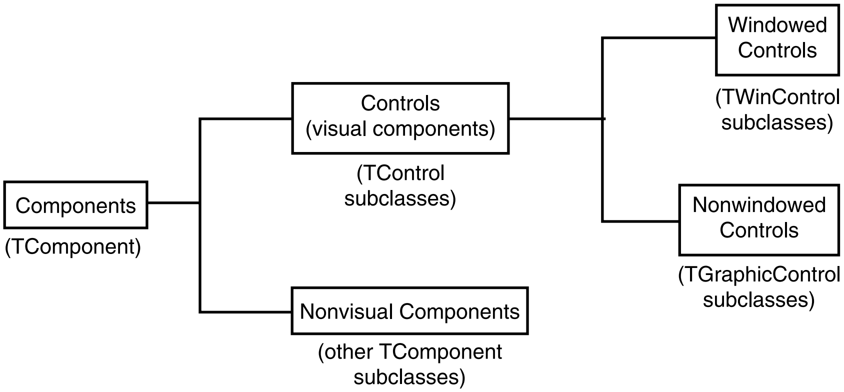
In Delphi 6, this distinction is underlined by the fact that the core non-visual components and classes of the library are part of the new RTL package, which is used by both VCL and CLX. Moreover, using this package in non-visual applications (for example, Web server programs) allows you to reduce the size of the files to deploy and load in memory considerably.

Traditional Sections of VCL

Delphi programmers use to refer to different sections of VCL with names Borland originally suggested in its documentation, and names that became common afterwards for different groups of components. Technically, components are subclasses of the `TComponent` class, which is one of the root classes of the hierarchy, as you can see in Figure 5.1. Actually the `TComponent` class inherits from the `TPersistent` class; the role of these two classes will be explained in the next section.

FIGURE 5.1:

A graphical representation of the main groups of components of VCL



Besides components, the library includes classes that inherit directly from `TObject` and from `TPersistent`. These classes are collectively known as *Objects* in portions of the documentation, a rather confusing name for me. These noncomponent classes are often used for values of properties or as utility classes used in code; not inheriting from `TComponent`, these classes cannot be used directly in visual programming.

NOTE To be more precise, noncomponent classes cannot be made available in the Component Palette and cannot be dropped directly into a form, but they can be visually managed with the Object Inspector, as subproperties of other properties or items of collections of various types. So even noncomponent classes are often easily used when interacting with the Form Designer.

The component classes can be further divided into two main groups: controls and nonvisual components. Controls groups all the classes that descend from `TControl`.

Controls have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different subspecifications, window-based or graphical, but I'll discuss them in more detail in the next chapter.

Nonvisual components are all the components that are not controls—all the classes that descend from `TComponent` but not from `TControl`. At design time, a nonvisual component appears on the form as an icon (optionally with a caption below it). At run time, some of these components may be visible (for example, the standard dialog boxes), and others are always invisible (for example, the database table component).

TIP You can simply move the mouse cursor over a control or component in the Form Designer to see a Tooltip with its name and class type (and, in Delphi 6, some extended information). You can also use an environment option, Show Component Captions, to see the name of a nonvisual component right under its icon.

The Structure of CLX

This is the traditional subdivision of VCL, which is very common for Delphi programmers. Even with the introduction of CLX and some new naming schemes, the traditional names will probably survive and merge into Delphi programmers' jargon.

Borland now refers to different portions of the CLX library using one terminology under Linux and a slightly different (and less clear) naming structure in Delphi. This is the subdivision of the Linux-compatible library:

BaseCLX forms the core of the class library, the topmost classes (such as `TComponent`), and several general utility classes (including lists, containers, collections, and streams). Compared to the corresponding classes of VCL, BaseCLX is largely unchanged and is highly portable between the Windows and Linux platforms. This chapter is largely devoted to exploring BaseCLX and the common VCL core classes.

VisualCLX is the collection of visual components, generally indicated as controls. This is the portion of the library that is more tightly related to the operating system: VisualCLX is implemented on top of the Qt library, available both on Windows and on Linux. Using VisualCLX allows for full portability of the visual portion of your application between Delphi on Windows and Kylix on Linux. However, most of the VisualCLX components have corresponding VCL controls, so that you can also easily move your code from one library to the other. I'll discuss VisualCLX and the controls of VCL in the next chapter.

DataCLX comprises all the database-related components of the library. Actually, DataCLX is the front end of the new dbExpress database engine included in Delphi 6 and Kylix. Delphi includes also the traditional BDE front end, dbGo, and InterBase Express (IBX). If we can consider all these components as part of DataCLX, only the dbExpress front end and IBX are portable between Windows and Linux. DataCLX includes also the ClientDataSet component, now indicated as MyBase, and other related classes. Delphi's data access components are discussed in Part III of the book.

NetCLX includes the Internet-related components, from the WebBroker framework, to the HTML producer components, from Indy (Internet Direct) to Internet Express, from the new Site Express to XML support. This part of the library is, again, highly portable between Windows and Linux. Internet support is discussed in the last part of the book.

VCL-Specific Sections of the Library

The preceding areas of the library are available, with the differences I've mentioned, on both Delphi and Kylix. In Delphi 6, however, there are other sections of VCL, which for one reason or another are specific to Windows only:

- The Delphi ActiveX (DAX) framework provides support for COM, OLE Automation, ActiveX, and other COM-related technologies. See Chapter 16 for more information on this area of Delphi.
- The Decision Cube components provide OLAP support but have ties with the BDE and haven't been updated recently. Decision Cube is not discussed in the book.

Finally, the default Delphi 6 installation includes some third-party components, such as TeeChart for business graphics and QuickReport for reporting. These components will be mentioned in the book but are not discussed in detail.

The *TPersistent* Class

The first core class of the Delphi library we'll look at is the `TPersistent` class, which is quite a strange one: it has very little code and almost no direct use, but it provides a foundation for the entire idea of visual programming. You can see the definition of the class in Listing 5.1.

Listing 5.1: The definition of the *TPersistent* class, from the Classes unit

```
{$M+}
TPersistent = class(TObject)
  private
    procedure AssignError(Source: TPersistent);
  protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFiler); virtual;
    function  GetOwner: TPersistent; dynamic;
  public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function  GetNamePath: string; dynamic;
  end;
```

As the name implies, this class handles persistency—that is, saving the value of an object to a file to be used later to re-create the object in the same state and with the same data. Persistency is a key element of visual programming. In fact (as we saw in Chapter 1) at design time in Delphi you manipulate actual objects, which are saved to DFM files and re-created at run time when the specific component container—form or data module—is created.

The streaming support, though, is not embedded in the `TPersistent` class but is provided by other classes, which target `TPersistent` and its descendants. In other words, you can "persist" with Delphi default streaming-only objects of classes inheriting from `TPersistent`. One of the reasons for this behavior lies in the fact that the class is compiled with a special option turned on, `{$M+}`. This flag activates the generation of extended RTTI information for the published portion of the class.

Delphi's streaming system, in fact, doesn't try to save the in-memory data of an object, which would be complex because of the many pointers to other memory locations, totally meaningless when the object would be reloaded. Instead, Delphi saves objects by listing the value of all of properties marked with a special keyword, `published`. When a property refers to another object, Delphi saves the name of the object or the entire object (with the same mechanism) depending on its type and relationship with the main object.

Of the methods of the `TPersistent` class, the only one you'll generally use is the `Assign` procedure, which can be used for copying the actual value of an object. In the library, this method is implemented by many noncomponent classes but by very few components. Actually, most subclasses reimplement the virtual protected `AssignTo` method, called by the default implementation of `Assign`.

NOTE Other methods include `DefineProperties`, used for customizing the streaming system and adding extra information (pseudo-properties), and the `GetOwner` and `GetNamePath` methods used by collections and other special classes to identify themselves to the Object Inspector.

The *published* Keyword

Along with the public, protected, and private access directives, you can use a fourth one, called published. For any published field, property, or method, the compiler generates extended RTTI information, so that Delphi's run time environment or a program can query a class for its published interface. For example, every Delphi component has a published interface that is used by the IDE, in particular the Object Inspector. A regular use of published fields is important when you write components. Usually, the published part of a component contains no fields or methods but properties and events.

When Delphi generates a form or data module, it places the definitions of its components and methods (the event handlers) in the first portion of its definition, before the public and private keywords. These fields and methods of the initial portion of the class are published. The default is published when no special keyword is added before an element of a component class.

To be more precise, published is the default keyword only if the class was compiled with the $M+ compiler directive or is descended from a class compiled with $M+. As this directive is used in the TPersistent class, most classes of VCL and all of the component classes default to published. However, noncomponent classes in Delphi (such as TStream and TList) are compiled with $M- and default to public visibility.

The methods assigned to any event should be published methods, and the fields corresponding to your components in the form should be published to be automatically connected with the objects described in the DFM file and created along with the form. (We'll see later in this chapter the details of this situation and the problems it generates.)

Accessing Published Fields and Methods

As I've mentioned, there are three different declarations that make sense in the published section of a class: fields, methods, and properties. I'll discuss properties in the section "Accessing Properties by Name," while here I'll introduce possible ways of interacting with fields and methods first. The TObject class, in fact, has three interesting methods for this area: MethodAddress, MethodName, and FieldAddress.

The first function, MethodAddress, returns the memory address of the compiled code (a sort of function pointer) of the method passed as parameter in a string. By assigning this method address to the Code field of a TMethod structure and assigning an object to the Data field, you can obtain a complete method pointer. At this point, to call the method you'll need to cast it to

Continued on next page

the proper method pointer type. This is a code fragment highlighting the key points of this technique:

```
var
  Method: TMethod;
  Evt: TNotifyEvent;
begin
  Method.Code := MethodAddress ('Button1Click');
  Method.Data := Self;
  Evt := TNotifyEvent(Method);
  Evt (Sender); // call the method
end;
```

Delphi uses similar code to assign an event handler when it loads a DFM file, as these files store the name of the methods used to handle the events, while the components actually store the method pointer. The second method, `MethodName`, does the opposite transformation, returning the name of the method at a given memory address. This can be used to obtain the name of an event handler, given its value, something Delphi does when streaming a component into a DFM file.

Finally, the `FieldAddress` method of `TObject` returns the memory location of a published field, given its name. This is used by Delphi to connect components created from the DFM files with the fields of their owner (for example, a form) having the same name.

Notice that these three methods are seldom used in "normal" programs but play a key role to make Delphi work as it actually does and are strictly related to the streaming system. You'll need to use these methods only when writing extremely dynamic programs or special-purpose wizards or other Delphi extensions.

Accessing Properties by Name

The Object Inspector displays a list of an object's published properties, even for components you've written. To do this, it relies on the RTTI information generated for published properties. Using some advanced techniques, an application can retrieve a list of the published properties of an object and use them.

Although this capability is not very well known, in Delphi it is possible to access properties by name simply by using the string with the name of the property and then retrieving its value. Access to the RTTI information of properties is provided through a group of undocumented subroutines, part of the TypInfo unit.

WARNING These subroutines have always been undocumented in past versions of Delphi, so that Borland remained free to change them. However, from Delphi 1 to Delphi 5, changes were actually very limited and related only to the data structures declared in TypInfo, not the functions provided by the unit. In Delphi 5 Borland actually added many more goodies, and a few "helper" routines, that are officially promoted (even if still not fully documented in the help file but only with comments provided in the unit).

Rather than explore the entire TypInfo unit here, we will look at only the minimal code required to access properties by name. Prior to Delphi 5 it was necessary to use the GetPropInfo function to retrieve a pointer to some internal property information and then apply one of the access functions, such as GetStrProp, to this pointer. You also had to check for the existence and the type of the property.

Delphi 5 introduced a new set of TypInfo routines, including the handy GetPropValue, which returns a variant with the value of the property or varNULL if the property doesn't exist. You simply pass to this function the object and a string with the property name. A further optional parameter allows you to choose the format for returning values of properties of the set type.

For example, we can call

```
ShowMessage (GetPropValue (Button1, 'Caption'));
```

This call has the same effect as calling ShowMessage, passing as parameter Button1.Caption. The only real difference is that this version of the code is much slower, since the compiler generally resolves normal access to properties in a more efficient way. The advantage of the run-time access is that you can make it very flexible, as in the following RunProp example (also available on the companion CD).

This program displays in a list box the value of a property of any type for each component of a form. The name of the property we are looking for is provided in an edit box. This makes the program very flexible. Besides the edit box and the list box, the form has a button to generate the output and some other components added only to test their properties. When you click the button, the following code is executed:

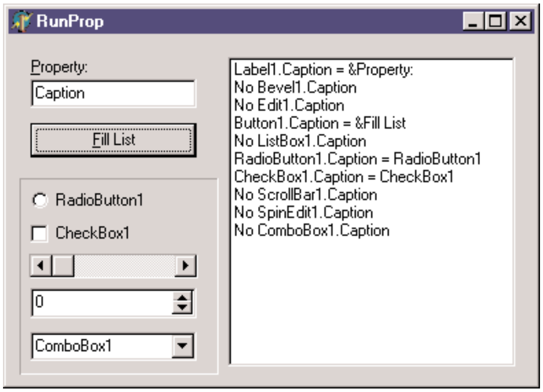
```
uses
  TypInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Value: Variant;
begin
  ListBox1.Clear;
```

```
for I := 0 to ComponentCount -1 do
begin
  if IsPublishedProp (Components[I], Edit1.Text) then
  begin
    Value := GetPropValue (Components[I], Edit1.Text);
    ListBox1.Items.Add (Components[I].Name + '.' + Edit1.Text + ' = ' + string (Value));
  end
  else
    ListBox1.Items.Add ('No ' + Components[I].Name + '.' + Edit1.Text);
```

You can see the effect of pressing the Fill List button while using the default *Caption* value in the edit box in Figure 5.2. You can try with any other property name. Numbers will be converted to strings by the variant conversion. Objects (such as the value of the `Font` property) will be displayed as memory addresses.

FIGURE 5.2:

The output of the RunProp example, which accesses properties by name at run time



```
Label1.Caption = &Property:
No Bevel1.Caption
No Edit1.Caption
Button1.Caption = &Fill List
No ListBox1.Caption
RadioButton1.Caption = RadioButton1
CheckBox1.Caption = CheckBox1
No ScrollBar1.Caption
No SpinEdit1.Caption
No ComboBox1.Caption
```

WARNING Do not use regularly the TypInfo unit instead of polymorphism and other property-access techniques. Use base-class property access first, or use the safe `as` typecast when required, and reserve RTTI access to properties as a very last resort. Using TypInfo techniques makes your code slower, more complex, and more prone to human error; in fact, it skips the compile-time type-checking.

The *TComponent* Class

If the `TPersistent` class is really more important than it seems at first sight, the key class at the heart of Delphi's component-based class library is `TComponent`, which inherits from

TPersistent (and from TObject). The TComponent class defines many core elements of components, but it is not as complex as you might think, as the base classes and the language already provide most of what's actually needed.

I won't explore all of the details of the TComponent class, some of which are more important for component designers than they are for component users. I'll just discuss ownership (which accounts for some public properties of the class) and the two published properties of the class, Name and Tag.

Ownership

One of the core features of the TComponent class is the definition of ownership. When a component is created, it can be assigned an owner component, which will be responsible for destroying it. So every component can have an owner and can also be the owner of other components. Several public methods and properties of the class are actually devoted to handling the *two sides* of ownership. Here is a list, extracted from the class declaration (in the Classes unit of VCL):

```
type
  TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
  public
    constructor Create(AOwner: TComponent); virtual;
    procedure DestroyComponents;
    function FindComponent(const AName: string): TComponent;
    procedure InsertComponent(AComponent: TComponent);
    procedure RemoveComponent(AComponent: TComponent);

    property Components[Index: Integer]: TComponent read GetComponent;
    property ComponentCount: Integer read GetComponentCount;
    property ComponentIndex: Integer
      read GetComponentIndex write SetComponentIndex;
    property Owner: TComponent read FOwner;
```

If you create a component giving it an owner, this will be added to the list of components (InsertComponent), which is accessible using the Components array property. The specific component has an Owner and knows its position in the owner components list, with the ComponentIndex property. Finally, the destructor of the owner will take care of the destruction of the object it owns, calling DestroyComponents. There are a few more protected methods involved, but this should give you the overall picture.

What is important to emphasize is that component ownership can solve a large part of the memory management problems of your applications, if used properly. If you always create components with an owner—the default operation if you use the visual designers of the IDE—you only need to remember to destroy these component containers when they are not needed anymore, and you can forget about the components they contain. For example, you delete a form to destroy all of the components it contains at once, which is a large simplification compared to having to remember to free each and every object individually.

The Components Array

The Components property can also be used to access one component owned by another—let's say, a form. This can be very handy (compared to using directly a specific component) for writing generic code, acting on all or many components at a time. For example, you can use the following code to add to a list box the names of all the components of a form (this code is actually part of the ChangeOwner example, presented in the next section):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do
    ListBox1.Items.Add (Components [I].Name);
end;
```

This code uses the ComponentCount property, which holds the total number of components owned by the current form, and the Components property, which is actually the list of the owned components. When you access a value from this list you get a value of the TComponent type. For this reason you can directly use only the properties common to all components, such as the Name property. To use properties specific to particular components, you have to use the proper type-downcast (as).

NOTE In Delphi, some components are also component containers: the GroupBox, Panel, PageControl, and, of course, Form components. When you use these controls, you can add other components inside them. In this case, the container is the parent of the components (as indicated by the Parent property), while the form is their owner (as indicated by the Owner property). You can use the Controls property of a form or group box to navigate the child controls, and you can use the Components property of the form to navigate all the owned components, regardless of their parent.

Using the Components property, we can always access each component of a form. If you need access to a specific component, however, instead of comparing each name with the name of the component you are looking for, you can let Delphi do this work, by using the FindComponent method of the form. This method simply scans the Components array looking for a name match. More information about the role of the Name property for a component is in a later section.

Changing the Owner

We have seen that almost every component has an owner. When a component is created at design time (or from the resulting DFM file), its owner will invariably be its form. When you create a component at run time, the owner is passed as a parameter to the `Create` constructor.

`Owner` is a read-only property, so you cannot change it. The owner is set at creation time and should generally not change during the lifetime of a component. To understand why you should not change the owner of a component at design time nor freely change its name, read the following discussion. Be warned, that the topic covered is not simple, so if you're only starting with Delphi, you might want to come back to this section at a later time.

To change the owner of a component, you can call the `InsertComponent` and `RemoveComponent` methods of the owner itself, passing the current component as parameter. Using these methods you can change a component's owner. However, you cannot apply them directly in an event handler of a form, as we attempt to do here:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveComponent (Button1);
  Form2.InsertComponent (Button1);
end;
```

This code produces a memory access violation, because when you call `RemoveComponent`, Delphi disconnects the component from the form field (`Button1`), setting it to `nil`. The solution is to write a procedure like this:

```
procedure ChangeOwner (Component, NewOwner: TComponent);
begin
  Component.Owner.RemoveComponent (Component);
  NewOwner.InsertComponent (Component);
end;
```

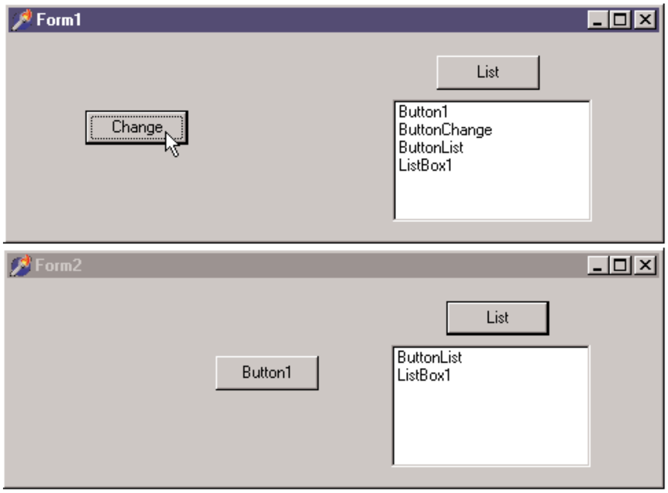
This method (extracted from the ChangeOwner example) changes the owner of the component. It is called along with the simpler code used to change the parent component; the two commands combined move the button *completely* to another form, changing its owner:

```
procedure TForm1.ButtonChangeClick(Sender: TObject);
begin
  if Assigned (Button1) then
  begin
    // change parent
    Button1.Parent := Form2;
    // change owner
    ChangeOwner (Button1, Form2);
  end;
end;
```

The method checks whether the Button1 field still refers to the control, because while moving the component, Delphi will set Button1 to nil. You can see the effect of this code in Figure 5.3.

FIGURE 5.3:

In the ChangeOwner example, clicking the Change button moves the Button1 component to the second form.



To demonstrate that the Owner of the Button1 component actually changes, I've added another feature to both forms. The List button fills the list box with the names of the components each form owns, using the procedure shown in the previous section. Click the two List buttons before and after moving the component, and you'll see what happens behind the scenes. As a final feature, the Button1 component has a simple handler for its OnClick event, to display the caption of the owner form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage ('My owner is ' + ((Sender as TButton).Owner as TForm).Caption);
end;
```

The *Name* Property

Every component in Delphi should have a name. The name must be unique within the owner component, which is generally the form into which you place the component. This means that an application can have two different forms, each with a component with the same name, although you might want to avoid this practice to prevent confusion. It is generally better to keep component names unique throughout an application.

Setting a proper value for the Name property is very important: If it's too long, you'll need to type a lot of code to use the object; if it's too short, you may confuse different objects.

Usually the name of a component has a prefix with the component type; this makes the code more readable and allows Delphi to group components in the combo box of the Object Inspector, where they are sorted by name. There are three important elements related to the Name property of the components:

- First, the value of the Name property is used to define the name of the object in the declaration of the form class. This is the name you're generally going to use in the code to refer to the object. For this reason, the value of the name property must be a legal Pascal identifier (it has to be without spaces and must start with a letter, not a number).
- Second, if you set the Name property of a control before changing its Caption or Text property, the new name is often copied to the caption. That is, if the name and the caption are identical, then changing the name will also change the caption.
- Third, Delphi uses the name of the component to create the default name of the methods related to its events. If you have a Button1 component, its default OnClick event handler will be called Button1Click, unless you specify a different name. If you later change the name of the component, Delphi will modify the names of the related methods accordingly. For example, if you change the name of the button to MyButton, the Button1Click method automatically becomes MyButtonClick.

As mentioned earlier, if you have a string with the name of a component, you can get its instance by calling the FindComponent of its owner, which returns nil in case the component is not found. For example, you can write

```
var
  Comp: TComponent;
begin
  Comp := FindComponent ('Button1');
  if Assigned (Comp) then
    with Comp as TButton do
      // some code...
```

NOTE Delphi includes also a FindGlobalComponent function, which finds a top-level component, basically a form or data module, that has a given name. To be precise, the FindGlobalComponent function calls one or more installed functions, so in theory you can modify the way the function works. However, as FindGlobalComponent is used by the streaming system, I strongly recommend against installing your own replacement functions. If you want to have a customized way to search for components on other containers, simply write a new function with a custom name.

Removing Form Fields

Every time you add a component to a form, Delphi adds an entry for it, along with some of its properties, to the DFM file. To the Pascal file, Delphi adds the corresponding field in the form class declaration. When the form is created, Delphi loads the DFM file and uses it to re-create all the components and set their properties back. Then it hooks the new object with the form field corresponding to its `Name` property.

For this reason, it is certainly possible to have a component without a name. If your application will not manipulate the component or modify it at run time, you can remove the component name from the Object Inspector. Examples are a static label with fixed text, or a menu item, or even more obviously, menu item separators. By blanking out the name, you'll remove the corresponding element from the form class declaration. This reduces the size of the form object (by only four bytes, the size of the object reference) and it reduces the DFM file by not including a useless string (the component name). Reducing the DFM also implies reducing the final EXE file size, even if only slightly.

WARNING If you blank out component names, just make sure to leave at least one named component of each class used on the form so that the smart linker will link in the required code for the class. If, as an example, you remove from a form all the fields referring to `TLabel` components, the Delphi linker will remove the implementation of the `TLabel` class from the executable file. The effect is that when the system loads the form at run time, it is unable to create an object of an unknown class and issues an error indicating that the class is not available.

You can also keep the component name and manually remove the corresponding field of the form class. Even if the component has no corresponding form field, it is created anyway, although using it (through the `FindComponent` method, for example) will be a little more difficult.

Hiding Form Fields

Many OOP purists complain that Delphi doesn't really follow the encapsulation rules, because all of the components of a form are mapped to public fields and can be accessed from other forms and units. Fields for components, in fact, are listed in the first unnamed section of a class declaration, which has a default visibility of published. However, Delphi does that only as a default to help beginners learn to use the Delphi visual development environment quickly. A programmer can follow a different approach and use properties and methods to operate on forms. The risk, however, is that another programmer of the same team might inadvertently bypass this approach, directly accessing the components if they are left in the published section. The solution, which many programmers don't know about, is to move the components to the private portion of the class declaration.

As an example, I've taken a very simple form with an edit box, a button, and a list box. When the edit box contains text and the user presses the button, the text is added to the list box. When the edit box is empty, the button is disabled. This is the simple code of the HideComp example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add (Edit1.Text);
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  Button1.Enabled := Length (Edit1.Text) <> 0;
end;
```

I've listed these methods only to show you that in the code of a form we usually refer to the available components, defining their interactions. For this reason it seems impossible to get rid of the fields corresponding to the component. However, what we can do is hide them, moving them from the default published section to the private section of the form class declaration:

```
TForm1 = class(TForm)
  procedure Button1Click(Sender: TObject);
  procedure Edit1Change(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  Button1: TButton;
  Edit1: TEdit;
  ListBox1: TListBox;
end;
```

Now if you run the program you'll get in trouble: The form will load fine, but because the private fields are not initialized, the events above will use `nil` object references. Delphi usually initializes the published fields of the form using the components created from the DFM file. What if we do it ourselves, with the following code?

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1 := FindComponent ('Button1') as TButton;
  Edit1 := FindComponent ('Edit1') as TEdit;
  ListBox1 := FindComponent ('ListBox1') as TListBox;
end;
```

It will *almost* work, but it generates a system error, similar to the one we discussed in the previous section. This time, the private declarations will cause the linker to link in the implementations of those classes, but the problem is that the streaming system needs to know the names of the classes in order to locate the class reference needed to construct the components while loading the DFM file.

The final touch we need is some registration code to tell Delphi at run time about the existence of the component classes we want to use. We should do this before the form is created, so I generally place this code in the initialization section of the unit:

```
initialization
  RegisterClasses ([TButton, TEdit, TListBox]);
```

Now the question is, is this really worth the effort? What we obtain is a higher degree of encapsulation, protecting the components of a form from other forms (and other programmers writing them). I have to say that replicating these steps for each and every form can be tedious, so I ended up writing a wizard to generate this code for me on the fly. The wizard is far from perfect, as it doesn't handle changes automatically, but it is usable. You can find it on my Web site, www.marcocantu.com, under the CanTools section. My simple wizard apart, for a large project built according to the principles of object-oriented programming, I recommend you consider this or a similar technique.

The Customizable Tag Property

The Tag property is a strange one, because it has no effect at all. It is merely an extra memory location, present in each component class, where you can store custom values. The kind of information stored and the way it is used are completely up to you.

It is often useful to have an extra memory location to attach information to a component without needing to define your component class. Technically, the Tag property stores a long integer so that, for example, you can store the entry number of an array or list that corresponds to an object. Using typecasting, you can store in the Tag property a pointer, an object, or anything else that is four bytes wide. This allows a programmer to associate virtually anything with a component using its tag. We'll see how to use this property in several examples in future chapters, including the ODMenu examples in Chapter 5.

Events

Now that I've introduced the TComponent class, there is one more element of Delphi we have to introduce. Delphi components, in fact, are programmed using "PME," properties, methods, and events. If methods and properties should be clear by now, events have not been fully introduced yet. The reason is that events don't imply a new language feature but are simply a standard coding technique. An event, in fact, is technically a property, with the only difference being that it refers to a method (a method pointer type, to be precise) instead of other types of data.

Events in Delphi

When a user does something with a component, such as clicking it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding operating system message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than operating system messages, and Delphi provides a number of extra inter-component messages.

From a theoretical point of view, an event is the result of a request sent to a component or control, which can respond to the message. Following this approach, to handle the click event of a button, we would need to subclass the `TButton` class and add the new event handler code inside the new class.

In practice, creating a new class for every component you want to use is too complex to be a reasonable solution. In Delphi, the event handler of a component usually is a method of the form that holds the component, not of the component itself. In other words, the component relies on its owner, the form, to handle its events. This technique is called *delegation*, and it is fundamental to the Delphi component-based model. This way, you don't have to modify the `TButton` class, unless you want to define a new type of component, but simply customize its owner to modify the behavior of the button.

Method Pointers

Events rely on a specific feature of the Object Pascal language: *method pointers*. A method pointer type is like a procedural type, but one that refers to a method. Technically, a method pointer type is a procedural type that has an implicit `Self` parameter. In other words, a variable of a procedural type stores the address of a function to call, provided it has a given set of parameters. A method pointer variable stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as `Self` inside the method body when the method code is called using this method pointer.

NOTE This explains the definition of Delphi's generic `TMethod` type, a record with a `Code` field and a `Data` field.

The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords of object at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

When you have declared a method pointer, such as the one above, you can declare a variable of this type and assign to it a compatible method—a method that has the same parameters—of another object.

When you add an OnClick event handler for a button, Delphi does exactly that. The button has a method pointer type property, named OnClick, and you can directly or indirectly assign to it a method of another object, such as a form. When a user clicks the button, this method is executed, even if you have defined it inside another class.

What follows is a sketch of the code actually used by Delphi to define the event handler of a button component and the related method of a form:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;
```

Now inside a procedure, you can write

```
MyButton.OnClick := Form1.Button1Click;
```

The only real difference between this code fragment and the code of VCL is that OnClick is a property name, and the actual data it refers to is called FOnClick. An event that shows up in the Events page of the Object Inspector, in fact, is nothing more than a property of a method pointer type. This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it.

Events Are Properties

Another important concept I've already mentioned is that events are properties. This means that to handle an event of a component, you assign a method to the corresponding event property. When you double-click an event in the Object Inspector, a new method is added to the owner form and assigned to the proper event property of the component.

This is why it is possible for several events to share the same event handler or change an event handler at run time. To use this feature, you don't need much knowledge of the language. In fact, when you select an event in the Object Inspector, you can press the arrow button on the right of the event name to see a drop-down list of "compatible" methods—a list of methods having the same method pointer type. Using the Object Inspector, it is easy to select the same method for the same event of different components or for different, compatible events of the same component.

As we've added some properties to the `TDate` class in Chapter 3, we can add one event. The event is going to be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the value of the date has changed. To define an event, we simply define a property corresponding to it, and we add some data to store the actual method pointer the event refers to. These are the new definitions added to the class, available in the DateEvt example:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FonChange write FOnChange;
    ...
  end;
```

The property definition is actually very simple. A user of this class can assign a new value to the property and, hence, to the `FOnChange` private field. The class doesn't assign a value to this `FOnChange` field; it is the user of the component who does the assignment. The `TDate` class simply calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned. The `DoChange` method (declared as a dynamic method as it is traditional with event firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

The DoChange method in turn is called every time one of the values changes, as in the following method:

```
procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
  // fire the event
  DoChange;
```

Now if we look at the program that uses this class, we can simplify its code considerably. First, we add a new custom method to the form class:

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

The code of this method simply updates the label with the current value of the Text property of the TDate object:

```
procedure TDateForm.DateChange;
begin
  LabelDate.Caption := TheDay.Text;
end;
```

This event handler is then installed in the FormCreate method:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (2001, 7, 4);
  LabelDate.Caption := TheDay.Text;
  // assign the event handler for future changes
  TheDay.OnChange := DateChange;
end;
```

Well, this seems like a lot of work. Was I lying when I told you that the event handler would save us some coding? No. Now, after we've added some code, we can completely forget about updating the label when we change some of the data of the object. Here, as an example, is the handler of the OnClick event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
end;
```

The same simplified code is present in many other event handlers. Once we have installed the event handler, we don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that we had to write some code at the beginning because this is not a component installed in Delphi but simply a class. With a component, you simply select the event handler in the Object Inspector and write a single line of code to update the label. That's all.

NOTE This is meant to be just a short introduction to defining events. A basic understanding of these features is important for every Delphi programmer. If your aim is to write new components, with complex events, you'll find a lot more information on all these topics in Chapter 11.

Lists and Container Classes

It is often important to handle groups of components or objects. Besides using standard arrays and dynamic arrays, there are a few classes of VCL that represent lists of other objects. These classes can be divided into three groups: simple lists, collections, and containers. The last group was introduced in Delphi 5 and has been further expanded in Delphi 6.

Lists and String Lists

Lists are represented by the generic list of objects, `TList`, and by the two lists of strings, `TStrings` and `TStringList`:

- `TList` defines a list of pointers, which can be used to store objects of any class. A `TList` is more flexible than a dynamic array, because it is expanded automatically, simply by adding new items to it. The advantage of dynamic arrays over a `TList`, instead, is that dynamic arrays allow you to indicate a specific type for contained objects and perform the proper compile-time type checking.
- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves, such as a list box.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with their own storage. You can use this class to define a list of strings in a program.

`TStringList` and `TStrings` objects have both a list of strings and a list of objects associated with the strings. This opens up a number of different uses for these classes. For example, you can use them for dictionaries of associated objects or to store bitmaps or other elements to be used in a list box.

The two classes of lists of strings also have ready-to-use methods to store or load their contents to or from a text file, SaveToFile and LoadFromFile. To loop through a list, you can use a simple for statement based on its index, as if the list were an array. All these lists have a number of methods and properties. You can operate on lists using the array notation ("[" and "]") both to read and to change elements. There is a Count property, as well as typical access methods, such as Add, Insert, Delete, Remove, and search methods (for example, IndexOf). In Delphi 6, the TList class has an Assign method that, besides copying the source data, can perform set operations on the two lists, including *and*, *or*, and *xor*.

To fill a string list with items and later check whether one is present, you can write code like this:

```
var
  sl: TStringList;
  idx: Integer;
begin
  sl := TStringList.Create;
  try
    sl.Add ('one');
    sl.Add ('two');
    sl.Add ('three');
    // later
    idx := sl.IndexOf ('two');
    if idx >= 0 then
      ShowMessage ('String found');
  finally
    sl.Free;
  end;
end;
```

Using Lists of Objects

We can write an example focusing on the use of the generic TList class. When you need a list of any kind of data, you can generally declare a TList object, fill it with the data, and then access the data while casting it to the proper type. The ListDemo example demonstrates just this. It also shows the pitfalls of this approach. Its form has a private variable, holding a list of dates:

```
private
  ListDate: TList;
```

This list object is created when the form itself is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ListDate := TList.Create;
end;
```

A button of the form adds a random date to the list (of course, I've included in the project the unit containing the date component built in the previous chapter):

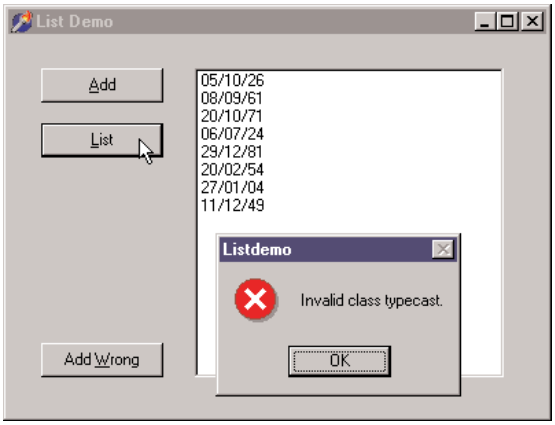
```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  ListDate.Add (TDate.Create (1900 + Random (200), 1 + Random (12),
    1 + Random (30)));
end;
```

When you extract the items from the list, you have to cast them back to the proper type, as in the following method, which is connected to the List button (you can see its effect in Figure 5.4):

```
procedure TForm1.ButtonListDateClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add ((TObject(ListDate [I]) as TDate).Text);
end;
```

FIGURE 5.4:

The list of dates shown by the ListDemo example



05/10/26
08/09/61
20/10/71
06/07/24
29/12/81
20/02/54
27/01/04
11/12/49

Invalid class typecast.

At the end of the code above, before we can do an as downcast, we first need to hard-cast the pointer returned by the TList into a TObject reference. This kind of expression can result in an invalid typecast exception, or it can generate a memory error when the pointer is not a reference to an object.

To demonstrate that things can indeed go wrong, I've added one more button, which adds a TButton object to the list:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  ListDate.Add (Sender);
end;
```

If you click this button and then update one of the lists, you'll get an error. Finally, remember that when you destroy a list of objects, you should remember to destroy all of the objects of the list first. The ListDemo program does this in the FormDestroy method of the form:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ListDate.Count - 1 do
    TObject(ListDate [I]).Free;
  ListDate.Free;
end;
```

Collections

The second group, collections, contains only two classes, TCollection and TCollectionItem. TCollection defines a homogeneous list of objects, which are owned by the collection class. The objects in the collection must be descendants of the TCollectionItem class. If you need a collection storing specific objects, you have to create both a subclass of TCollection and a matching subclass of TCollectionItem.

Collections are used to specify values of properties of components. It is very unusual to work with collections for storing your own objects, so I won't discuss them here.

Container Classes

Delphi 5 introduced a new series of container classes, defined in the Contnrs unit. Delphi 6 extends these classes by adding hashed associative lists, as discussed in the following section. The container classes extend the TList classes by adding the idea of ownership and by defining specific extraction rules (mimicking stacks and queues) or sorting capabilities.

The basic difference between TList and the new TObjectList class, for example, is that the latter is defined as a list of TObject objects, not a list of pointers. Even more important, however, is the fact that if the object list has the OwnsObjects property set to True, it automatically deletes an object when it is replaced by another one and deletes each object when the list itself is destroyed. Here's a list of all the new container classes:

- The `TObjectList` class I've already described represents a list of objects, eventually owned by the list itself.
- The inherited class `TComponentList` represents a list of components, with full support for destruction notification (an important safety feature when two components are connected using their properties; that is, when a component is the value of a property of another component).
- The `TClassList` class is a list of class references. It inherits from `TList` and requires no destruction.
- The classes `TStack` and `TObjectStack` represent lists of pointers and objects, from which you can only extract elements starting from the last one you've inserted. A stack follows the LIFO order (Last In, First Out). The typical methods of a stack are Push for insertion, Pop for extraction, and Peek to preview the first item without removing it. You can still use all the methods of the base class, `TList`.
- The classes `TQueue` and `TObjectQueue` represent lists of pointers and objects, from which you always remove the *first* item you've inserted (FIFO: first in, first out). The methods of these classes are the same as those of the stack classes but behave differently.

WARNING Unlike the `TObjectList`, the `TObjectStack` and the `TObjectQueue` do not own the inserted objects and will not destroy those objects left in the data structure when it is destroyed. You can simply `Pop` all the items, destroy them once you're finished using them, and then destroy the container.

To demonstrate the use of these classes, I've modified the earlier ListDate example into the new Contain example on the CD. First, I changed the type of the `ListDate` variable to `TObjectList`. In the `FormCreate` method, I've modified the list creation to the following code, which activates the list ownership:

```
ListDate := TObjectList.Create (True);
```

At this point, we can simplify the destruction code, as applying `Free` to the list will automatically free the dates it holds.

I've also added to the program a stack and a queue object, filling each of them with numbers. One of the form's two buttons displays a list of the numbers in each container, and the other removes the last item (displayed in a message box):

```
procedure TForm1.btnQueueClick(Sender: TObject);
var
  I: Integer;
```

```
begin
  ListBox1.Clear;
  for I := 0 to Stack.Count - 1 do begin
    ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));
    Queue.Push(Queue.Pop);
  end;
  ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));
end;
```

By pressing the two buttons, you can see that calling Pop for each container returns the last item. The difference is that the TQueue class inserts elements at the beginning, and the TStack class inserts them at the end.

Hashed Associative Lists

After whetting our appetite in Delphi 5, Borland has pushed the idea of container classes a little further in Delphi 6, introducing a new set of lists, particularly TBucketList and TObjectBucketList. These two lists are associative, which means they have a key and an actual entry. The key is used to identify the items and search for them. To add an item, you call the Add method, with two parameters, the key and the actual data. When you use the Find method, you pass the key and retrieve the data. The same effect is achieved by using the Data array property, passing the key as parameter.

These lists are also based on a hash system. The lists create an internal array of items, called buckets, each having a sub-list of actual elements of the list. As you add an item, its key value is used to compute the *hash* value, which determines the bucket to add the item to. When searching the item, the hash is computed again, and the list immediately grabs the sublist containing the item, searching for it there. This makes for very fast insertion and searches, but only if the hash algorithm distributes the items evenly among the various buckets and if there are enough different entries in the array. In fact, when many elements can be in the same bucket, searching gets slower.

For this reason, as you create the TObjectBucketList you can specify the number of entries for the list, using the parameter of the constructor, choosing a value between 2 and 256. The value of the bucket is determined by taking the first byte of the pointer (or number) passed as key and doing an and operation with a number corresponding to the entries.

NOTE I don't find this algorithm very convincing for a hash system, but replacing it with your own implies only overriding the BucketFor virtual function and eventually changing the number of entries in the array, by setting a different value for the BucketCount property.

Another interesting feature, not available for lists, is the ForEach method, which allows you to execute a given function on each item contained in the list. You pass to the ForEach method a pointer to data of your own and a procedure, which receives four parameters, including your custom pointer, each key and object of the list, and a Boolean parameter you can set to False to stop the execution. In other words, these are the two signatures:

```
type
  TBucketProc = procedure(AInfo, AItem, AData: Pointer;
    out AContinue: Boolean);

function TCustomBucketList.ForEach(AProc: TBucketProc;
  AInfo: Pointer): Boolean;
```

NOTE Besides these containers, Delphi includes also a THashedStringList class, which inherits from TStringList. This class has no direct relationship with the hashed lists and is even defined in a different unit,IniFiles The hashed string list has two associated hash tables (of type TStringHash), which are completely refreshed every time the content of the string list changes. So this class is useful only for reading a large set of fixed strings, not for handling a list of strings changing often over time. On the other hand, the TStringHash support class seems to be quite useful in general cases, and has a good algorithm for computing the hash value of a string.

Type-Safe Containers and Lists

Containers and lists have a problem: They are not type-safe, as I've shown in both examples by adding a button object to a list of dates. To ensure that the data in a list is homogenous, you can check the type of the data you extract before you insert it, but as an extra safety measure you might also want to check the type of the data while extracting it. However, adding run-time type checking slows down a program and is risky—a programmer might fail to check the type in some cases.

To solve both problems, you can create specific list classes for given data types and fashion the code from the existing TList or TObjectList classes (or another container class). There are two approaches to accomplish this:

- Derive a new class from the list class and customize the Add method and the access methods, which relate to the Items property. This is also the approach used by Borland for the container classes, which all derive from TList.

NOTE Delphi container classes use static overrides to perform simple type conveniences (parameters and function results of the desired type). Static overrides are not the same as polymorphism; someone using a container class via a TList variable will not be calling the container's specialized functions. Static override is a simple and effective technique, but it has one very important restriction: The methods in the descendent should not do anything beyond simple type-casting, because you aren't guaranteed that the descendent methods will be called. The list might be accessed and manipulated using the ancestor methods as much as by the descendent methods, so their actual operations must be identical. The only difference is the type used in the descendent methods, which allows you to avoid extra typecasting.

- Create a brand-new class that contains a TList object, and map the methods of the new class to the internal list using proper type checking. This approach defines a wrapper class, a class that "wraps" around an existing one to provide a different or limited access to its methods (in our case, to perform a type conversion).

I've implemented both solutions in the DateList example, which defines lists of TDate objects. In the code that follows, you'll find the declaration of the two classes, the inheritance-based TDateListI class and the wrapper class TDateListW.

```
type
// inheritance-based
TDateListI = class (TObjectList)
protected
  procedure SetObject (Index: Integer; Item: TDate);
  function GetObject (Index: Integer): TDate;
public
  function Add (Obj: TDate): Integer;
  procedure Insert (Index: Integer; Obj: TDate);
  property Objects [Index: Integer]: TDate
    read GetObject write SetObject; default;
end;

// wrapper based
TDateListW = class(TObject)
private
  FList: TObjectList;
  function GetObject (Index: Integer): TDate;
    procedure SetObject (Index: Integer; Obj: TDate);
  function GetCount: Integer;
public
  constructor Create;
  destructor Destroy; override;
  function Add (Obj: TDate): Integer;
  function Remove (Obj: TDate): Integer;
  function IndexOf (Obj: TDate): Integer;
  property Count: Integer read GetCount;
  property Objects [Index: Integer]: TDate
    read GetObject write SetObject; default;
end;
```

Obviously, the first class is simpler to write—it has fewer methods, and they simply call the inherited ones. The good thing is that a TDateListI object can be passed to parameters expecting a TList. The problem is that the code that manipulates an instance of this list via a generic TList variable will not be calling the specialized methods, because they are not virtual and might end up adding to the list objects of other data types.

Instead, if you decide not to use inheritance, you end up writing a lot of code, because you need to reproduce each and every one of the original `TList` methods, simply calling the methods of the internal `FList` object. The drawback is that the `TDateListW` class is not type compatible with `TList`, which limits its usefulness. It can't be passed as parameter to methods expecting a `TList`.

Both of these approaches provide good type checking. After you've created an instance of one of these list classes, you can add only objects of the appropriate type, and the objects you extract will naturally be of the correct type. This is demonstrated by the DateList example. This program has a few buttons, a combo box to let a user choose which of the lists to show, and a list box to show the actual values of the list. The program stretches the lists by trying to add a button to the list of `TDate` objects. To add an object of a different type to the `TDateListI` list, we can simply convert the list to its base class, `TList`. This might accidentally happen if you pass the list as a parameter to a method that expects a base class object. In contrast, for the `TDateListW` list to fail we must explicitly cast the object to `TDate` before inserting it, something a programmer should never do:

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);
begin
  ListW.Add (TDate(TButton.Create (nil)));
  TList(ListI).Add (TButton.Create (nil));
  UpdateList;
end;
```

The `UpdateList` call triggers an exception, displayed directly in the list box, because I've used an `as` typecast in the custom list classes. A wise programmer should never write the above code. To summarize, writing a custom list for a specific type makes a program much more robust. Writing a wrapper list instead of one that's based on inheritance tends to be a little safer, although it requires more coding.

NOTE Instead of rewriting wrapper-style list classes for different types, you can use my List Template Wizard, available on my Web site, **`www.marcocantu.com`**.

Streaming

Another core area of the Delphi class library is its support for streaming, which includes file management, memory, sockets, and other sources of information arranged in a sequence. The idea of streaming is that you move along the data while reading it, much like the traditional `read` and `write` functions used by the Pascal language (and discussed in Chapter 12 of *Essential Pascal*, available on the companion CD).

The *TStream* Class

The VCL defines the abstract TStream class and several subclasses. The parent class, TStream, has just a few properties, and you'll never create an instance of it, but it has an interesting list of methods you'll generally use when working with derived stream classes.

The TStream class defines two properties, Size and Position. All stream objects have a specific size (which generally grows if you write something after the end of the stream), and you must specify a position within the stream where you want to either read or write information.

Reading and writing bytes depends on the actual stream class you are using, but in both cases you don't need to know much more than the size of the stream and your relative position in the stream to read or write data. In fact, that's one of the advantages of using streams. The basic interface remains the same whether you're manipulating a disk file, a binary large object (BLOB) field, or a long sequence of bytes in memory.

In addition to the Size and Position properties, the TStream class also defines several important methods, most of which are virtual and abstract. (In other words, the TStream class doesn't define what these methods do; therefore, derived classes are responsible for implementing them.) Some of these methods are important only in the context of reading or writing components within a stream (for instance, ReadComponent and WriteComponent), but some are useful in other contexts, too. In Listing 5.2, you can find the declaration of the TStream class, extracted from the Classes unit.

Listing 5.2: The public portion of the definition of the *TStream* class

```
TStream = class(TObject)
public
  // read and write a buffer
  function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
  function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
  procedure ReadBuffer(var Buffer; Count: Longint);
  procedure WriteBuffer(const Buffer; Count: Longint);

  // move to a specific position
  function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;
  function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
    overload; virtual;

  // copy the stream
  function CopyFrom(Source: TStream; Count: Int64): Int64;

  // read or write a component
  function ReadComponent(Instance: TComponent): TComponent;
  function ReadComponentRes(Instance: TComponent): TComponent;
  procedure WriteComponent(Instance: TComponent);
  procedure WriteComponentRes(const ResName: string; Instance: TComponent);
```

```
    procedure WriteDescendent(Instance, Ancestor: TComponent);
    procedure WriteDescendentRes(
      const ResName: string; Instance, Ancestor: TComponent);
    procedure WriteResourceHeader(const ResName: string; out FixupInfo: Integer);
    procedure FixupResourceHeader(FixupInfo: Integer);
    procedure ReadResHeader;

    // properties
    property Position: Int64 read GetPosition write SetPosition;
    property Size: Int64 read GetSize write SetSize64;
  end;
```

The basic use of a string involves calling the ReadBuffer and WriteBuffer methods, which are very powerful but not terribly easy to use. The first parameter, in fact, is an untyped buffer in which you can pass the variable to save from or load to. For example, you can save into a file a number (in binary format) and a string, with this code:

```
var
  stream: TStream;
  n: integer;
  str: string;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test', fmCreate);
  stream.WriteBuffer (n, sizeOf(integer));
  stream.WriteBuffer (str[1], Length (str));
  stream.Free;
```

A totally alternative approach is to let specific components save or load data to and from streams. Many VCL classes define a LoadFromStream or a SaveToStream method, including TStrings, TStringList, TBlobField, TMemoField, TIcon, and TBitmap.

Specific Stream Classes

Creating a TStream instance makes no sense, because this class is abstract and provides no direct support for saving data. Instead, you can use one of the derived classes to load data from or store it to an actual file, a BLOB field, a socket, or a memory block. Use TFileStream when you want to work with a file, passing the filename and some file access options to the Create method. Use TMemoryStream to manipulate a stream in memory and not an actual file.

Several units define TStream-derived classes. In the Classes unit are the following classes:

- THandleStream defines a stream that manipulates a disk file represented by a Windows file handle.

- TFileStream defines a stream that manipulates a disk file (a file that exists on a local or network disk) represented by a filename. It inherits from THandleStream.
- TCustomMemoryStream is the base class for streams stored in memory but is not used directly.
- TMemoryStream defines a stream that manipulates a sequence of bytes in memory. It inherits from TCustomMemoryStream.
- TStringStream provides a simple way for associating a stream to a string in memory, so that you can access the string with the TStream interface and also copy the string to and from another stream.
- TResourceStream defines a stream that manipulates a sequence of bytes in memory, and provides read-only access to resource data linked into the executable file of an application (an example of these resource data are the DFM files). It inherits from TCustomMemoryStream.

Stream classes defined in other units include

- TBlobStream defines a stream that provides simple access to database BLOB fields. There are similar BLOB streams for other database access technologies rather than the BDE.
- TOleStream defines a stream for reading and writing information over the interface for streaming provided by an OLE object.
- TWinSocketStream provides streaming support for a socket connection.

Using File Streams

Creating and using a file stream can be as simple as creating a variable of a type that descends from TStream and calling components methods to load content from the file:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
  begin
    S := TFileStream.Create (OpenDialog1.FileName, fmOpenRead);
    try
      Memo1.Lines.LoadFromStream (S);
    finally
      S.Free;
    end;
  end;
end;
```

As you can see in this code, the Create method for file streams has two parameters: the name of the file and a flag indicating the requested access mode. In this case, we want to read the file, so we used the fmOpenRead flag (other available flags are documented in the Delphi help).

NOTE Of the different modes, the most important are fmShareDenyWrite, which you'll use when you're simply reading data from a shared file, and fmShareExclusive, which you'll use when you're writing data to a shared file.

A big advantage of streams over other file access techniques is that they're very interchangeable, so you can work with memory streams and then save them to a file, or you can perform the opposite operations. This might be a way to improve the speed of a file-intensive program. Here is a snippet of code, a file-copying function, to give you another idea of how you can use streams:

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create (SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create (TargetName, fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom (Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end
  finally
    Stream1.Free;
  end
end;
```

Another important use of streams is to handle database BLOB fields or other large fields directly. In fact, you can export such data to a stream or read it from one by simply calling the SaveToStream and LoadFromStream methods of the TBlobField class.

The *TReader* and *TWriter* Classes

By themselves, the stream classes of VCL don't provide much support for reading or writing data. In fact, stream classes don't implement much beyond simply reading and writing blocks of data. If you want to load or save specific data types in a stream (and don't want to perform a great deal of typecasting), you can use the TReader and TWriter classes, which derive from the generic TFiler class.

Basically, the TReader and TWriter classes exist to simplify loading and saving stream data according to its type, and not just as a sequence of bytes. To do this, TWriter embeds special signatures into the stream that specify the type for each object's data. Conversely, the TReader class reads these signatures from the stream, creates the appropriate objects, and then initializes those objects using the subsequent data from the stream.

For example, I could have written out a number and a string to a stream by writing:

```
var
  stream: TStream;
  n: integer;
  str: string;
  w: TWriter;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test.txt', fmCreate);
  w := TWriter.Create (stream, 1024);
  w.WriteInteger (n);
  w.WriteString (str);
  w.Free;
  stream.Free;
```

This time the actual file will include also the extra signature characters, so that I can read back this file only by using a TReader object. For this reason, using the TReader and TWriter is generally confined to components streaming and is seldom applied in general file management.

Streams and Persistency

In Delphi, streams play a considerable role for persistency. For this reason, many methods of TStream relate to saving and loading a component and its subcomponents. For example, you can store a form in a stream by writing

```
stream.WriteComponent(Form1);
```

If you examine the structure of a Delphi DFM file, you'll discover that it's really just a resource file that contains a custom format resource. Inside this resource, you'll find the component information for the form or data module and for each of the components it contains. As you would expect, the stream classes provide two methods to read and write this custom resource data for components: WriteComponentRes to store the data, and ReadComponentRes to load it.

For your experiment in memory (not involving actual DFM files), though, using WriteComponent is generally better suited. After you create a memory stream and save the current form to it, the problem is how to display it. This can be accomplished by transforming the binary representation of forms to a textual representation. Even though the Delphi

IDE, since version 5, can save DFM files in text format, the representation used internally for the compiled code is invariably a binary format.

The form conversion can be accomplished by the IDE, generally with the View as Text command of the form designer, and in other ways. There is also a command-line utility, `CONVERT.EXE`, found in the Delphi `Bin` directory. Within your own code, the standard way to obtain a conversion is to call the specific methods of VCL. There are four functions for converting to and from the internal object format obtained by the `WriteComponent` method:

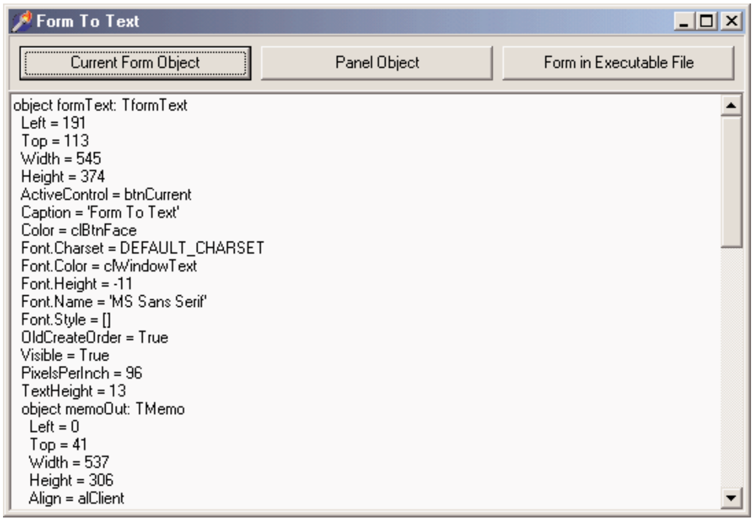
```
procedure ObjectBinaryToText(Input, Output: TStream); overload;
procedure ObjectBinaryToText(Input, Output: TStream;
  var OriginalFormat: TStreamOriginalFormat); overload;
procedure ObjectTextToBinary(Input, Output: TStream); overload;
procedure ObjectTextToBinary(Input, Output: TStream;
  var OriginalFormat: TStreamOriginalFormat); overload;
```

Four different functions, with the same parameters and names containing the name *Resource* instead of *Binary* (as in `ObjectResourceToText`), convert the resource format obtained by `WriteComponentRes`. A final method, `TestStreamFormat`, indicates whether a DFM is storing a binary or textual representation.

In the FormToText program, I've used the `ObjectBinaryToText` method to copy the binary definition of a form into another stream, and then I've displayed the resulting stream in a memo, as you can see in Figure 5.5. This is the code of the two methods involved:

FIGURE 5.5:

The textual description of a form component, displayed inside itself by the FormToText example



```
object formText: TformText
  Left = 191
  Top = 113
  Width = 545
  Height = 374
  ActiveControl = btnCurrent
  Caption = 'Form To Text'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = True
  Visible = True
  PixelsPerInch = 96
  TextHeight = 13
  object memoOut: TMemo
    Left = 0
    Top = 41
    Width = 537
    Height = 306
    Align = alClient
```

```
procedure TformText.btnCurrentClick(Sender: TObject);
var
  MemStr: TStream;
begin
  MemStr := TMemoryStream.Create;
  try
    MemStr.WriteComponent (Self);
    ConvertAndShow (MemStr);
  finally
    MemStr.Free
  end;
end;

procedure TformText.ConvertAndShow (aStream: TStream);
var
  ConvStream: TStream;
begin
  aStream.Position := 0;
  ConvStream := TMemoryStream.Create;
  try
    ObjectBinaryToText (aStream, ConvStream);
    ConvStream.Position := 0;
    MemoOut.Lines.LoadFromStream (ConvStream);
  finally
    ConvStream.Free
  end;
end;
```

Notice that by repeatedly clicking the Current Form Object button you'll get more and more text, and the text of the memo is included in the stream. After a few times, the entire operation will get extremely slow, so that the program seems to be hung up. In this code, we start to see some of the flexibility of using streams—we can write a generic procedure we can use to convert any stream.

NOTE It's important to stress that after you've written data to a stream, you must explicitly seek back to the beginning (or set the `Position` property to 0) before you can use the stream further, unless you want to append data to the stream, of course.

Another button, labeled Panel Object, shows the textual representation of a specific component, the panel, passing the component to the `WriteComponent` method. The third button, Form in Executable File, does a different operation. Instead of streaming an existing object in memory, it loads in a TResourceStream object the design-time representation of the form—that is, its DFM file—from the corresponding resource embedded in the executable file:

```
procedure TformText.btnResourceClick(Sender: TObject);
var
  ResStr: TResourceStream;
begin
  ResStr := TResourceStream.Create(hInstance, 'TFORMTEXT', RT_RCDATA);
  try
    ConvertAndShow (ResStr);
  finally
    ResStr.Free
  end;
end;
```

By clicking the buttons in sequence (or modifying the form of the program) you can compare the form saved in the DFM file to the current run-time object.

Writing a Custom Stream Class

Besides using the existing stream classes, Delphi programmers can write their own stream classes, and use them in place of the existing ones. To accomplish this, you need only specify how a generic block of raw data is saved and loaded, and VCL will be able to use your new class wherever you call for it. You may not need to create a brand-new stream class for working with a new type of media, but only need to customize an existing stream. In that case, all you have to do is write the proper read and write methods.

As an example, I created a class to encode and decode a generic file stream. Although this example is limited by its use of a totally dumb encoding mechanism, it fully integrates with VCL and works properly. The new stream class simply declares the two core reading and writing methods and has a property that stores a key.

```
type
  TEncodedStream = class (TFileStream)
  private
    FKey: Char;
  public
    constructor Create(const FileName: string; Mode: Word);
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    property Key: Char read FKey write FKey;
  end;
```

Continued on next page

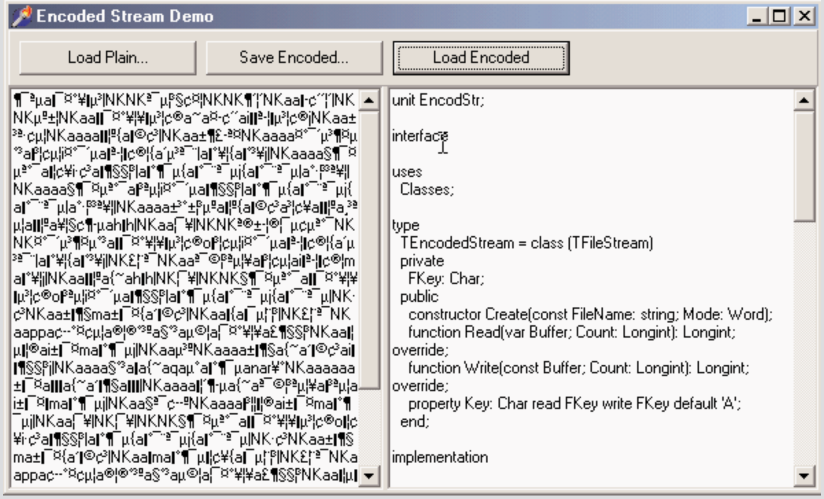
The value of the key is simply added to each of the bytes saved to a file, and subtracted when the data is read. Here is the complete code of the Write and Read methods, which uses pointers quite heavily:

```
constructor TEncodedStream.Create( const FileName: string; Mode: Word);
begin
  inherited Create (FileName, Mode);
  FKey := 'A'; // default
end;

function TEncodedStream.Write(const Buffer; Count: Longint): Longint;
var
  pBuf, pEnc: PChar;
  I, EncVal: Integer;
begin
  // allocate memory for the encoded buffer
  GetMem (pEnc, Count);
  try
    // use the buffer as an array of characters
    pBuf := PChar (@Buffer);
    // for every character of the buffer
    for I := 0 to Count - 1 do
    begin
      // encode the value and store it
      EncVal := ( Ord (pBuf[I]) + Ord(Key) ) mod 256;
      pEnc [I] := Chr (EncVal);
    end;
    // write the encoded buffer to the file
    Result := inherited Write (pEnc^, Count);
  finally
    FreeMem (pEnc, Count);
  end;
end;

function TEncodedStream.Read(var Buffer; Count: Longint): Longint;
var
  pBuf, pEnc: PChar;
  I, CountRead, EncVal: Integer;
begin
  // allocate memory for the encoded buffer
  GetMem (pEnc, Count);
  try
```

Continued on next page

```
    // read the encoded buffer from the file
    CountRead := inherited Read (pEnc^, Count);
    // use the output buffer as a string
    pBuf := PChar (@Buffer);
    // for every character actually read
    for I := 0 to CountRead - 1 do
    begin
      // decode the value and store it
      EncVal := ( Ord (pEnc[I]) - Ord(Key) ) mod 256;
      pBuf [I] := Chr (EncVal);
    end;
  finally
    FreeMem (pEnc, Count);
  end;
  // return the number of characters read
  Result := CountRead;
end;
```

The comments in this rather complex code should help you understand the details. Now that we have an encoded stream, we can try to use it in a demo program, which is called EncDemo. The form of this program has two memo components and three buttons, as you can see in the graphic below. The first button loads a plain text file in the first memo; the second button saves the text of this first memo in an encoded file; and the last button reloads the encoded file into the second memo, decoding it. In this example, after encoding the file, I've reloaded it in the first memo as a plain text file on the left, which of course is unreadable.



```
unit EncodStr;

interface

uses
  Classes;

type
  TEncodedStream = class (TFileStream)
  private
    FKey: Char;
  public
    constructor Create(const FileName: string; Mode: Word);
    function Read(var Buffer; Count: Longint): Longint;
override;
    function Write(const Buffer; Count: Longint): Longint;
override;
    property Key: Char read FKey write FKey default 'A';
  end;

implementation
```

Continued on next page

Since we have the encoded stream class available, the code of this program is very similar to that of any other program using streams. For example, here is the method used to save the encoded file (you can compare its code to that of earlier examples based on streams):

```
procedure TFormEncode.BtnSaveEncodedClick(Sender: TObject);
var
  EncStr: TEncodedStream;
begin
  if SaveDialog1.Execute then
  begin
    EncStr := TEncodedStream.Create(SaveDialog1.Filename, fmCreate);
    try
      Memo1.Lines.SaveToStream (EncStr);
    finally
      EncStr.Free;
    end;
  end;
end;
```

Summarizing the Core VCL and BaseCLX Units

We've spent most of the space of this chapter discussing the classes of a single unit of the library, Classes. This unit is certainly important, but it is not the only core unit of the library (although there aren't many others). In this section, I'm providing an overview of these units and their content.

The Classes Unit

The Classes unit is at the heart of both VCL and CLX libraries, and though it sees many internal changes from the last version of Delphi, there is little new for the average users. (Most changes are related to modified IDE integration and are meant for expert component writers.)

Here is a list of what you can find in the Classes unit, a unit that every Delphi programmer should spend some time with:

- Many enumerated types, the standard method pointer types (including `TNotifyEvent`), and many exception classes.

- Core library classes, including TPersistent and TComponent but also TBasicAction and TBasicActionLink.
- List classes, including TList, TThreadList (a thread-safe version of the list), TInterfaceList (a list of interfaces, used internally), TCollection, TCollectionItem, TOwnedCollection (which is simply a collection with an owner), TStrings, and TStringList.
- All the stream classes I discussed in the previous section but won't list here again. There are also the TFiler, TReader, and TWriter classes and a TParser class used internally for DFM parsing.
- Utility classes, such as TBits for binary manipulation and a few utility routines (for example, point and rectangle constructors, and string list manipulation routines such as LineStart and ExtractStrings). There are also many registration classes, to notify the system of the existence of components, classes, special utility functions you can replace, and much more.
- The TDataModule class, a simple object container alternative to a form. Data modules can contain only nonvisual components and are generally used in database and Web applications.

NOTE In past versions of Delphi, the TDataModule class was defined in the Forms unit; now it has been moved to the Classes unit. This was done to eliminate the code overhead of the GUI classes from non-visual applications (for example, Web server modules) and to better separate non-portable Windows code from OS-independent classes, such as TDataModule. Other changes relate to the data modules, for example, to allow the creation of Web applications with multiple data modules, something not possible in Delphi 5.

- New interface-related classes, such as TInterfacedPersistent, aimed at providing further support for interfaces. This particular class allows Delphi code to hold onto a reference to a TPersistent object or any descendent implementing interfaces, and is a core element of the new support for interfaced objects in the Object Inspector (see Chapter 11 for an example).
- The new TRecall class, used to maintain a temporary copy of an object, particularly useful for graphical-based resources.
- The new TClassFinder class used for finding a registered class instead of the FindClass method.
- The TThread class, which provides the core to operating system–independent support for multithreaded applications.

Other Core Units

Other units that are part of the RTL package are not directly used by typical Delphi programmers as often as Classes. Here is a list:

- The TypInfo unit includes support for Accessing RTTI information for published properties, as we've seen in the section "Accessing Properties by Name."
- The SyncObjs unit contains a few generic classes for thread synchronization.

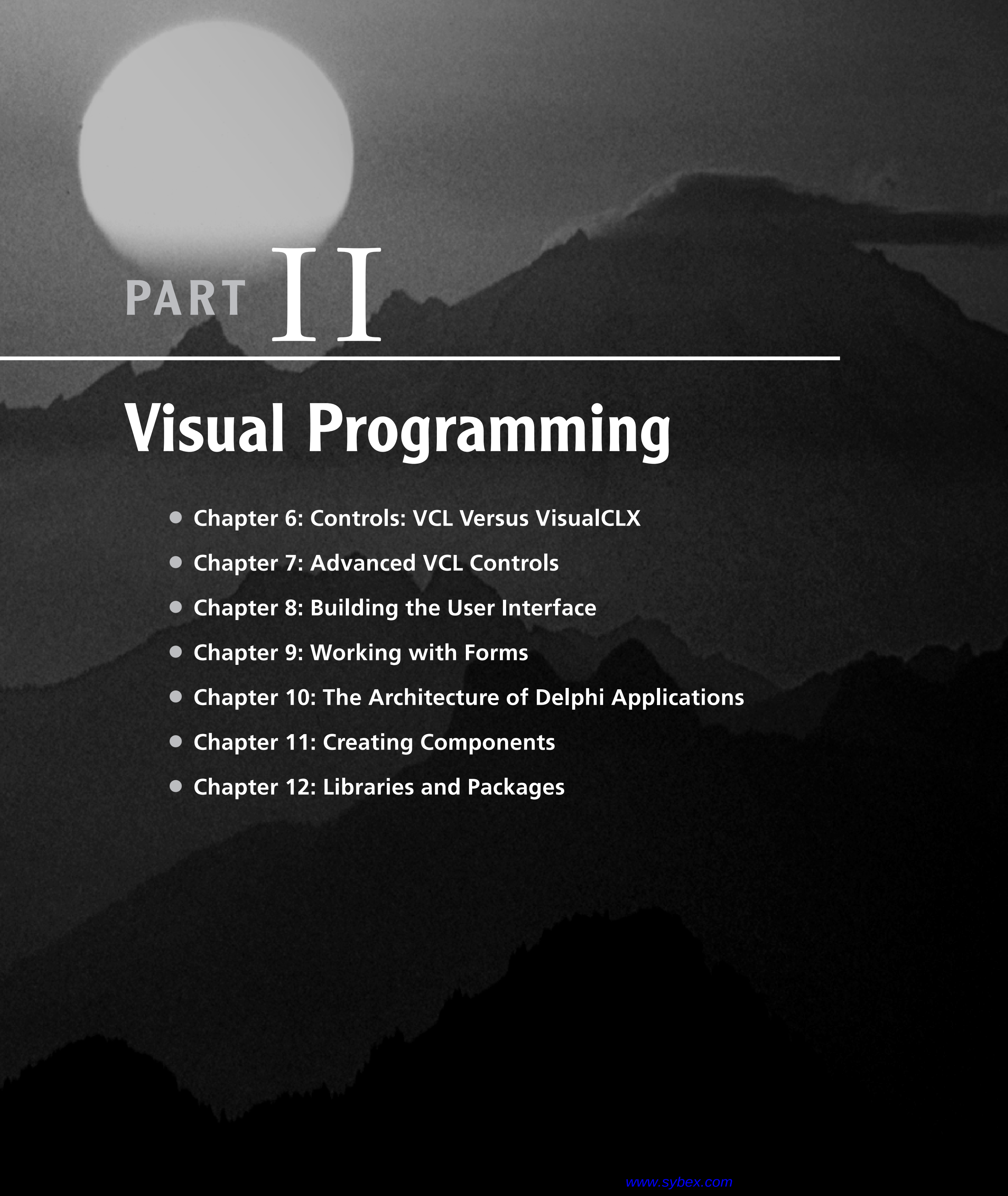
Of course, the RTL package also includes the units with functions and procedures discussed in the preceding chapter, such as Math, SysUtils, Variants, VarUtils, StrUtils, DateUtils, and so on.

What's Next?

As we have seen in this chapter, the Delphi class library has a few root classes that play a considerable role and that you should learn to leverage to the maximum possible extent. Some programmers tend to become expert on the components they use every day, and this is important, but without understanding the core classes (and ideas such as ownership and streaming), you'll have a tough time grasping the full power of Delphi.

Of course, in this book, we also need to discuss visual and database classes, which I will do in the next chapter. Now that we've seen all the base elements of Delphi (language, RTL, core classes), we are ready to discuss the development of real applications with this tool.

Part II of the book, which starts with the next chapter, is fully devoted to examples of the use of the various components, particularly visual components with the development of the user interface. We'll start with the advanced use of traditional controls and menus, discuss the actions architecture, cover the `TForm` class, and then examine toolbars, status bars, dialog boxes, and MDI applications in later chapters. Then we'll move to the development of database applications in Part III of the book.



PART II

Visual Programming

CHAPTER 6

Controls: VCL Versus VisualCLX

- VCL versus VisualCLX
- TControl, TWinControl, and TWidgetControl
- An overview of the standard components
- Basic and advanced menu construction
- Modifying the system menu
- Graphics in menus and list boxes
- OwnerDraw and styles

Now that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the base elements of component library, we are ready to delve into the second part of the book: the use of components and the development of the user interface of applications. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

Delphi comes with a large number of ready-to-use components. I won't describe every component in detail, examining each of its properties and methods; if you need this information, you can find it in the Help system. The aim of Part II of this book is to show you how to use some of the advanced features offered by the Delphi predefined components to build applications and to discuss specific programming techniques.

I'll start with a comparison of the VCL and VisualCLX libraries available in Delphi 6 and a coverage of the core classes (particularly `TControl`). Then I'll try to list all the various visual components you have, because choosing the right basic controls is often a way to get into a project faster.

VCL versus VisualCLX

As we've seen in the last chapter, Delphi 6 introduces the new CLX library alongside the traditional VCL library. There are certainly many differences, even in the use of the RTL and code library classes, between developing programs specifically for Windows or with a cross-platform attitude, but the user interface portion is where differences are most striking.

The visual portion of VCL is a wrapper of the Window API. It includes wrappers of the native Windows controls (like buttons and edit boxes), of the common controls (like tree views and list views), plus a bunch of native Delphi controls bound to the Windows concept of a window. There is also a `TCanvas` class that wraps the basic graphic calls, so you can easily paint on the surface of a window.

VisualCLX, the visual portion of CLX, is a wrapper of the Qt (pronounced "cute") library. It includes wrappers of the native Qt widgets, which range from basic to advanced controls, very similar to Windows' own standard and common controls. It includes also painting support using another, similar, `TCanvas` class. Qt is a C++ class library, developed by Trolltech (`www.trolltech.com`), a Norwegian company with a strong relationship with Borland.

On Linux, Qt is one of the de facto standard user-interface libraries and is the basis of the KDE desktop environment. On Windows, Qt provides an alternative to the use of the native APIs. In fact, unlike VCL, which provides a wrapper to the native controls, Qt provides an alternate implementation to those controls. This allows programs to be truly portable, as there are no hidden differences created by the operating system (and that the operating system vendor can introduce behind the scenes). It also allows us to avoid an extra layer; CLX on top of Qt on top of Windows native controls suggests three layers, but in fact there are two layers in each solution (CLX controls on top of Qt, VCL controls on top of Windows).

NOTE Distributing Qt applications on Windows implies the distribution of the Qt library itself (something you can generally take for granted on the Linux platform). Distributing the Qt libraries with a professional application (as opposed to an open source project) generally implies paying a license to Trolltech. If you use Delphi or Kylix to build Qt applications, however, Borland has already paid the license to Trolltech for you. However, you must use the CLX classes wrapping Qt: If you use the Qt classes directly, you apparently still owe the license to Qt, even when using Delphi or Kylix.

Technically, there are huge differences behind the scenes between a native Windows application built with VCL and a portable Qt program developed with VisualCLX. Suffice to say that at the low level, Windows uses API function calls and messages to communicate with controls, while Qt uses class methods and direct method callbacks and has no internal messages. Technically, the Qt classes offer a high-level object-oriented architecture, while the Windows API is still bound to its C legacy and a message-based system dated 1985 (when Windows was released). VCL offers an object-oriented abstraction on top of a low-level API, while VisualCLX remaps an already high-level interface into a more *familiar* class library.

NOTE To be honest, Microsoft has apparently reached the point of starting to abandon the traditional low-level Windows API for a native high-level class library, part of the dotNet architecture. Of course, this change won't happen overnight, but new high-level user-interface technologies might be introduced only in dotNet. Actually, dotNet consists of multiple technologies, including a virtual machine or runtime interpreter, a low-level nonvisual RTL, and a class framework for visual stuff (partially overlapping with VCL. If having a new visual class library on top of the Windows API might be of little use to programmers already using a modern class library (like VCL) other areas of dotNet would be of interest to Delphi programmers. So far, Borland has released no official statement regarding possible support for the dotNet byte code and virtual machine, or other areas of the future Microsoft operating system offering.

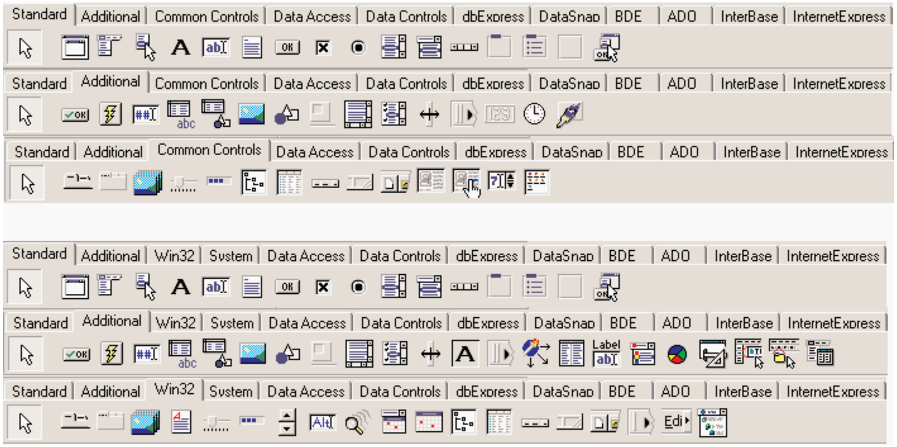
Having a familiar class library on top of a totally new platform is the advantage for Delphi programmers of using VisualCLX on Linux. This implies that the two class libraries, CLX and VCL, are very similar for their users, even if they are very different internally, as I mentioned. From the outside, a button is an object of the `TButton` class for both libraries, and it has more or less the same set of methods, properties, and events. In many occasions, you can recompile your existing programs for the new class library in a matter of minutes, if they don't map directly to low-level APIs.

Delphi 6 Dual Libraries Support

Delphi 6 has full support for both libraries at design time and at run time. As you start developing a new application, you can use the File ➢ New Application command to create a new VCL-based program and File ➢ New CLX Application for a new CLX-based program. After giving one of these two commands, Delphi's IDE will create a VCL or CLX design-time form and update the Component Palette so that it displays only the visual components compatible with the type of application you've selected (see Figure 6.1 for a comparison). In fact, you cannot place a VCL button into a CLX form, and you cannot even mix forms of the libraries within a single executable file. In other words, the user interface of every application must be built using exclusively one of the two libraries, which (aside from the technical implications) actually makes a lot of sense to me.

FIGURE 6.1:

A comparison of the first three pages of the Component Palette for a CXL-based application (above) and a VCL-based application (below)


Standard | Additional | Common Controls | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | InternetExpress
Standard | Additional | Common Controls | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | InternetExpress
Standard | Additional | Common Controls | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | InternetExpress
Standard | Additional | Win32 | System | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | InternetExpress
Standard | Additional | Win32 | System | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | InternetExpress
Standard | Additional | Win32 | System | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | InternetExpress

If you haven't already done so, I suggest you to try experimenting with the creation of a CLX application, looking at the available controls and trying to use them. You'll find very few differences in the use of the components, and if you have been using Delphi for some time, you'll probably be immediately adept with CLX.

Same Classes, Different Units

One of the cornerstones of the source-code compatibility between CLX and VCL code is that fact that similar classes in the two libraries have exactly the same class name. Each library has a class called `TButton` representing a push button; the methods and properties are so similar, this code will work with both libraries:

```
with TButton.Create (Self) do
begin
  SetBounds (20, 20, 80, 35);
```

```
    Caption := 'New';
    Parent := Self;
  end;
```

The two TButton classes have the same name, and this is possible because they are saved in two different units, called StdCtrls and QStdCtrls. Of course, you cannot have the two components available at design time in the palette, as the Delphi IDE can register only components with unique names. The entire VisualCLX library is defined by units corresponding to the VCL units, but with the letter *Q* as a prefix—so there is a QForms unit, a QDialogs unit, a QGraphics unit, and so on. There are also a few peculiar ones, such as the QStyle unit, that have no correspondence in VCL.

Notice that there are no compile settings or other hidden techniques to distinguish between the two libraries; what matters is the set of units referenced in the code. Remember that these references must be consistent, as you cannot mix visual controls of the two libraries in a single form and not even in a single program.

DFM and XFM

As you create a form at design time, this is saved to a form definition file. Traditional VCL applications use the DFM extension, which stands for Delphi form module. CLX applications use the XFM extension, which stands for cross-platform (i.e., *X*) form modules. The actual format of DFM or XFM files, which can be based on a textual or binary representation, is identical. A form module is the result of streaming the form and its components, and the two libraries share the streaming code, so they produce a fairly similar effect.

So the reason for having two different extensions doesn't lie in internal compiler tricks or incompatible formats. It is merely an indication to programmers and to the IDE of the type of components you should expect to find within that definition (as this indication is *not* included in the file itself).

If you want to convert a DFM file into an XFM file, you can simply rename the file. However, expect to find some differences in the properties, events, and available components, so that reopening the form definition for a different library will probably cause quite a few warnings.

TIP

Apparently Delphi's IDE chooses the active library only by looking at the extension of the form module, ignoring the references in the `uses` statements. For this reason, do change the extension if you plan using CLX. On Kylix, a different extension is pretty useless, because any form is opened in the IDE as a CLX form, regardless of the extension. On Linux, there is only the Qt-based CLX library, which is both the cross-platform *and* the native library.

As an example, I've built two simple identical applications, LibComp and QLibComp (available on this book's CD-ROM), with only a few components and a single event handler. Listing 6.1 presents the textual form definitions for two applications, built using the same steps in the Delphi 6 IDE, after choosing a CLX or VCL application. I've marked out differences in bold; as you can see, there are very few, most relating to the form and its font. The `OldCreateOrder` is a legacy property, used for compatibility with Delphi 3 and older code; standard colors have different names; and CLX saves the scrollbars' ranges.

Listing 6.1: An XFM file (left) and an equivalent DFM file (right)

```
object Form1: TForm1              object Form1: TForm1
  Left = 192                        Left = 192
  Top = 107                         Top = 107
  Width = 350                       Width = 350
  Height = 210                      Height = 210
  Caption = 'QLibComp'              Caption = 'LibComp'
  Color = clBackground              Color = clBtnFace
  VertScrollBar.Range = 161         Font.Charset = DEFAULT_CHARSET
  HorzScrollBar.Range = 297         Font.Color = clWindowText
                                    Font.Height = -11
                                    Font.Name = 'MS Sans Serif'
                                    Font.Style = []
  TextHeight = 13                   TextHeight = 13
  TextWidth = 6                     OldCreateOrder = False
  PixelsPerInch = 96                PixelsPerInch = 96
  object Button1: TButton           object Button1: TButton
    Left = 56                         Left = 56
    Top = 64                          Top = 64
    Width = 75                        Width = 75
    Height = 25                       Height = 25
    Caption = 'Add'                   Caption = 'Add'
    TabOrder = 0                      TabOrder = 0
    OnClick = Button1Click            OnClick = Button1Click
  end                               end
  object Edit1: TEdit                object Edit1: TEdit
    Left = 40                         Left = 40
    Top = 32                          Top = 32
    Width = 105                       Width = 105
    Height = 21                       Height = 21
    TabOrder = 1                      TabOrder = 1
    Text = 'my name'                  Text = 'my name'
  end                               end
  object ListBox1: TListBox         object ListBox1: TListBox
    Left = 176                        Left = 176
    Top = 32                          Top = 32
    Width = 121                       Width = 121
    Height = 129                      Height = 129
    Rows = 3                          ItemHeight = 13
    Items.Strings = (                 Items.Strings = (
   'marco'
   'john'
   'helen')
  TabOrder = 2
 end
end
```

```
   'marco'
   'john'
   'helen')
  TabOrder = 2
 end
end
```

uses Statements

By looking at the source code of the two examples, the differences are even less relevant, as they simply relate to the uses statements. The form of the CLX application has the following initial code:

```
unit QLibCompForm;
interface
uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;
```

The form of the VCL program has the traditional uses statement:

```
unit LibCompForm;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
```

The code of the class and of the only event handler is absolutely identical. Of course, the classic compiler directive {$R *.dfm} is replaced by {$R *.xfm} in the CLX version of the program.

Disabling the Dual Library Help Support

In Delphi 6, when you press the F1 key in the editor asking for help on a routine, class, or method of the Delphi library, you'll usually get a choice between the VCL and CLX declarations of the same feature. You'll need to make a choice to proceed to the related help page, which can be quite annoying after a while (especially as the two pages are often identical).

If you don't care about CLX and are planning to use only VCL (or vice versa), you can disable this alternative by choosing the Help ➢ Customize command, removing everything with *CLX* in the name from Contents, Index, and Link, and saving the project. Then restart the Delphi IDE, and the Help engine won't bother asking you about CLX any more. Of course, don't forget to add those help files again in case you decide to start using CLX.

Choosing a Visual Library

Because you have two different user interface libraries available in Delphi 6, you'll have to choose one for each *visual* application. You must evaluate multiple criteria to come to the proper decision, which isn't always easy.

The first criterion is portability. If running your program on Windows and on Linux, with the same user interface, is a major concern to you, using CLX will probably make your life simpler and let you keep a single source code file with very limited IFDEFs. The same applies if you consider Linux to be (or possibly become) your key platform. Instead, if most of your users are on Windows and you just want to extend your offering with a Linux version, you might want to keep a dual VCL/CLX system. This probably implies two different sets of source code files, or too many for IFDEFs.

In fact, another criterion is the native look-and-feel. By using CLX on Windows, some of the controls will behave slightly differently than users will expect—at least expert users. For a simple user interface (edits, buttons, grids), this probably won't matter much, but if you have many tree view and list view controls, the differences will be quite clear. On the other hand, with CLX you'll be able to let your users select a look-and-feel of their choice, different from the basic Windows look, and use it consistently across platforms.

Using native controls implies also that as soon as you get a new version of the Windows operating system, your application will (probably) adapt to it. This is good for the user, but might cause you a lot of headaches in case of incompatibilities. Differences in the Microsoft common controls library over the last few years have been a major source of frustration for Windows programmers in general, including Delphi programmers.

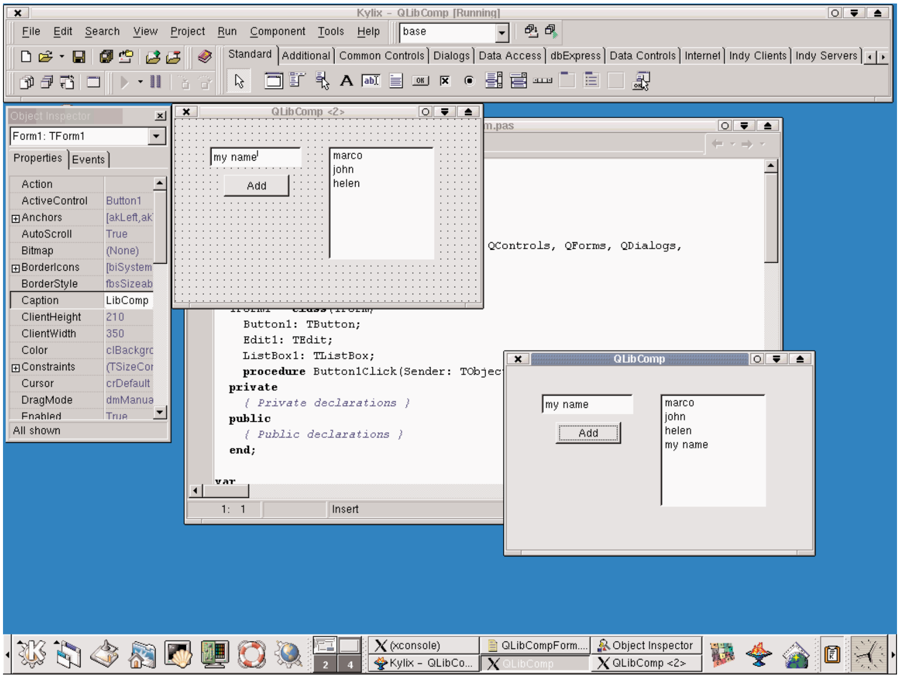
Another criterion is the deployment: If you use CLX, you'll have to ship your Windows program with the Qt libraries, which are not commonly available on Windows systems.

Finally, I've done a little testing, and it seems that the speed of VCL and CLX applications is similar. I've tried creating a thousand components, showing them on screen, and the speed differences are few, with a slight advantage for the VCL-based solution. You can try them out with the LibSpeed and QLibSpeed applications on the companion CD.

Running It on Linux

So the real issue of choosing the library resolves to the importance of Linux for you and your users. What is very important to notice is that, if you create a CLX application, you'll be able to recompile it unchanged (with the exact source code) with Kylix producing a native Linux application.

As an example, I've recompiled the QLibComp example introduced earlier, and you can see it running in Figure 6.2, where you can also see the Kylix IDE in action on a KDE 2 SuSE system.



Action	
ActiveControl	Button1
⊞Anchors	[akLeft,ak
AutoScroll	True
Bitmap	(None)
⊞BorderIcons	[biSystem
BorderStyle	fbsSizeab
Caption	LibComp
ClientHeight	210
ClientWidth	350
Color	clBackgrc
⊞Constraints	(TSizeCor
Cursor	crDefault
DragMode	dmManua
Enabled	True

```
QControls, QForms, QDialogs,

    Button1: TButton;
    Edit1: TEdit;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObjec
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

FIGURE 6.2: An application written with CLX can be directly recompiled under Linux with Kylix (displayed in the background).

Conditional Compilation for Libraries

If you want to keep a single source code file but compile with VCL on Windows and CXL on Linux, you can use platform-specific symbols (such as `$IFDEF LINUX`) to distinguish the two situations in case of conditional compilation. But what if you want to be able to compile a portion of code for both libraries on Windows?

You can either define a symbol of your own, and use conditional compilation, or (at times) test for the presence of identifiers that exist only in VCL or CLX only, as in:

```
{$IF Declared(QForms)}
  ...CLX-specific code
{$IFEND}
```

Converting Existing Applications

Besides starting with new CLX applications, you might want to convert some of your existing VCL applications to the new class library. There are a series of operations you have to do, without any specific help from the Delphi IDE:

- You'll have to rename the DFM file as XFM and update all of the `{$R *.DFM}` statements as `{$R *.XFM}`.
- You'll have to update all of the `uses` statements of your program (in the units and project files) to refer to the CLX units instead of the VCL units. Notice that by missing even a few, you'll bump into trouble when running your application.

TIP To prevent a CLX application from compiling if it contains references to VCL units, you can move the VCL units to a different directory under `lib` and avoid including this folder in your search path. This way, eventual leftover references to VCL units will cause a "Unit not found" error.

Table 6.1 is a comparison of the names of the visual VCL and CLX units, excluding the database portion and some rarely referenced units:

TABLE 6.1: Names of Equivalent VCL and CLX Units

VCL	CLX
ActnList	QActnList
Buttons	QButtons
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	QConsts
Controls	QControls
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdCtrls	QStdCtrls

You might also convert references to Windows and Messages into references to the Qt unit. Some Windows data structures are now also available in the Types unit (see Chapter 4, “The Run-Time Library,” for details), so you might have to add it to your CLX programs. Notice, however, that the QTypes unit is not the CLX version of VCL’s Types unit; these two units are totally unrelated.

WARNING Watch out for your `uses` statements! If you happen to compile a project that includes a CLX form, but fail to update the project unit, leaving a reference to the VCL Forms unit there, your program will run but stop immediately. The reason is that no VCL form was created, so the program terminated right away. In other cases, trying to create a CLX form within a VCL application will cause run-time errors. Finally, the Delphi IDE might inappropriately add references to `uses` statements of the wrong library, so you end up with a single `uses` statement referring to the same unit for both, but only the second of the two will be effective. This rarely prevents the program from compiling, but you won’t be able to run it.

The VclToClx Helper Tool

As a helper in converting some of my own programs, I’ve written a simple unit-replacement tool, called VclToClx and available with its complete source code in the `Tools` folder of the book CD and on my Web site.

The program converts unit names, based on a configuration file, and fixes the DFM issue, by renaming the DFM files to XFM and fixing the references in the source code. The program is quite naive, as it doesn’t really parse the source code, but simply looks for the occurrences of the unit names followed by a comma or semicolon, as happens in a `uses` statement. It also requires that the unit name is preceded by a space, but of course you can modify the program to look for a comma. Don’t skip this extra test; otherwise the Forms unit will be turned to QForms, but the QForms unit will be converted again to QQForms!

TControl and Derived Classes

In the preceding chapter, I discussed the base classes of the Delphi library, focusing particularly on the `TComponent` class. One of the most important subclasses of `TComponent` is `TControl`, which corresponds to visual components. This base class is available both in CLX and VCL and defines general concepts, such as the position and the size of the control, the parent control hosting it, and more. For an actual implementation, though, you have to refer to its two subclasses. In VCL these are TWinControl and TGraphicControl; in CLX they are TWidgetControl and TGraphicControl. Here are their key features:

- *Window-based controls* (also called *windowed controls*) are visual components based on an operating-system window. A `TWinControl` in VCL has a window handle, a number referring to an internal Windows structure. A `TWidgetControl` in CLX has a Qt handle, a reference to the internal Qt object. From a user perspective, windowed controls can receive the input focus, and some of them can contain other controls. This is the biggest group of components in the Delphi library. We can further divide windowed controls in two groups: wrappers of native controls of Windows or Qt, and custom controls, which generally inherit from `TCustomControl`.

- *Graphical controls* (also called *nonwindowed controls*) are visual components that are not based on an operating-system window. Therefore, they have no handle, cannot receive the focus, and cannot contain other controls. These controls inherit from `TGraphicControl` and are painted by their parent form, which sends them mouse-related and other events. Examples of nonwindowed controls are the Label and SpeedButton components. There are just a few controls in this group, which were critical to minimizing the use of system resources in the early days of Delphi (on 16-bit Windows). Using graphical controls to save Windows resources is still quite useful on Win9*x*/Me, which has pushed the system limits higher but hasn't fully gotten rid of them (unlike Windows NT/2000).

A Short History of Windows Controls

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window that has a specific behavior and some styles and is capable of responding to specific messages. These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components. (Actually, Microsoft's third step was its ActiveX technology, which is now followed by the dotNet framework, which is more or less at the level of the VCL controls.)

Windows 3.1 had six kinds of predefined controls, which were generally used in dialog boxes. Still used in Win32, they are buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Windows 95 added new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. Win32 developers can use the standard common controls provided by the system, and Delphi developers have the further advantage of having corresponding easy-to-use components.

As we have seen, Qt offers to CLX comparable basic and common controls, and even if there are internal differences, the Delphi libraries exposing those controls provide wrappers that can minimize those differences. VCL, in fact, literally wraps Windows predefined controls in some of its basic components. A Delphi wrapper class—for example, `TEdit`—simply surfaces the capabilities of the underlying Windows control, making it easier to use. However, Delphi adds nothing to the capabilities of this control. In Windows 95/98, an edit or memo control has a physical limit of 32 KB of text, and this limit is retained by the Delphi component.

Continued on next page

Why hasn't Borland overcome this limit? Why can't we change the color of a button? Simply because by replacing a Windows control with a custom version, we would lose the close connection with the operating system. Suppose Microsoft improves some of the controls in the next version of Windows. If we use our own version of the component, the application we build won't have the new features. By using controls that are based on the operating-system capabilities, instead, our programs have the opportunity to migrate through different versions of the OS and retain all the features provided by the specific version. This doesn't apply to the use of Qt, of course, but you have the advantage of being able to have an identical application based on the same source code running on Linux.

Note that wrapping an existing Windows or Qt control is an effective way of reusing code and also helps reduce the size of your compiled program. Implementing yet another button control from scratch requires custom code in your application, while a wrapper around the OS-supplied button control requires less code and makes use of system code shared by many applications.

Parent and Controls

The `Parent` property of a control indicates which other control is responsible for displaying it. When you drop a component into a form in the Form Designer, the form will become both parent and owner of the new control. But if you drop the component inside a Panel, ScrollBox, or any other *container* component, this will become its parent, while the form will still be the owner of the control.

When you create the control at run time, you'll need to set the owner (using the `Create` constructor parameter); but you must also set the `Parent` property, or the control won't be visible.

Like the `Owner` property, the `Parent` property has an inverse. The `Controls` array, in fact, lists all of the controls parented by the current one, numbered from `0` to `ControlsCount - 1`. You can scan this property to operate on all of the controls hosted by another one, eventually using a recursive method that operates on the controls parented by each subcontrol.

Properties Related to Control Size and Position

Some of the properties introduced by `TControl` and common to all controls are those related to size and position. The position of a control is determined by its `Left` and `Top` properties, its size by the `Height` and `Width` properties. Technically, all components have a position, because when you reopen an existing form at design time, you want to be able to see the icons for the nonvisual components in exactly the position where you've placed them. This position is visible in the form file.

TIP As you change any of the positional or size properties, you end up calling the single SetBounds method. So any time you need to change two or more of these properties at once, calling SetBounds directly will speed up the program. Another method, BoundsRect, returns the rectangle bounding of the control and corresponds to accessing those four properties.

An important feature of the position of a component is that, like any other coordinate, it always relates to the client area of its parent component (indicated by its Parent property). For a form, the client area is the surface included within its borders (excluding the borders themselves). It would have been messy to work in screen coordinates, although there are some ready-to-use methods that convert the coordinates between the form and the screen and vice versa.

Note, however, that the coordinates of a control are always relative to the parent control, such as a form or another *container* component. If you place a panel in a form, and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In fact, in this case, the parent component of the button is the panel.

Activation and Visibility Properties

There are two basic properties you can use to let the user activate or hide a component. The simpler is the Enabled property. When a component is disabled (when Enabled is set to False), usually some visual hint indicates this state to the user. At design time, the "disabled" property does not always have an effect, but at run time, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding Hide method or by setting its Visible property to False. Be aware, however, that reading the status of the Visible property does not tell you whether the control is actually visible. In fact, if the container of a control is hidden, even if the control is set to Visible, you cannot see it. For this reason, there is another property, Showing, which is a run-time and read-only property. You can read the value of Showing to know whether the control is really visible to the user; that is, if it is visible, its parent control is also visible, the parent control of the parent control is also visible, and so on.

Fonts

Two properties often used to customize the user interface of a component are Color and Font. Several properties are related to the color. The Color property itself usually refers to the background color of the component. Also, there is a Color property for fonts and many other graphic elements. Many components also have a ParentColor and a ParentFont property, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the Font property of the form itself.

When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Delphi allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users' computers.

If your program uses a font that your user doesn't have, Windows will select some other font to use in its place. A program's carefully formatted output can be ruined by the font substitution. For this reason, you should probably rely only on standard Windows fonts (such as MS Sans Serif, System, Arial, Times New Roman, and so on).

Colors

There are various ways to set the value of a color. The type of this property is `TColor`. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include clBlue, clSilver, clWhite, clGreen, clRed, and many others.

TIP Delphi 6 adds four new standard colors: clMoneyGreen, clSkyBlue, clCream, and clMedGray.

As a better alternative, you can use one of the colors used by the system to denote the status of given elements. These sets of colors are different in VCL and CLX. VCL includes predefined Windows colors such as the background of a window (clWindow), the color of the text of a highlighted menu (clHightlightText), the active caption (clActiveCaption), and the ubiquitous button face color (clBtnFace).

CLX includes a different and incompatible set of system colors, including clBackground, which is the standard color of a form; clBase, used by edit boxes and other visual controls; clActiveForeground, the foreground color for active controls; and clDisabledBase, the background color for disabled text controls. All the color constants mentioned here are listed in VCL and CLX Help files under the "TColor type" topic.

Another option is to specify a `TColor` as a number (a 4-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low three bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value $00FF0000 corresponds to a pure blue color, the value $0000FF00 to green, the value $000000FF to red, the value $00000000 to black, and the value $00FFFFFF to white. By specifying intermediate values, you can obtain any of 16 million possible colors.

Instead of specifying these hexadecimal values directly, you should use the Windows `RGB` function, which has three parameters, all ranging from 0 to 255. The first indicates the amount of red, the second the amount of green, and the last the amount of blue. Using the `RGB` function makes programs generally more readable than using a single hexadecimal constant. Actually, RGB is *almost* a Windows API function. It is defined by the Windows-related units and not by Delphi units, but a similar function does not exist in the Windows API. In C, there is a macro that has the same name and effect, so this is a welcome addition to the Pascal interface to Windows. RGB is not available on CLX, so I've written my own version as:

```
function RGB (red, green, blue: Byte): Cardinal;
begin
  Result := blue + green * 256 + red * 256 * 256;
end;
```

The highest-order byte of the TColor type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.) Regarding palettes and color matching, note that Windows sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, Windows uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors. In 16-color (VGA) adapters and at higher resolutions, you often end up seeing strange patterns of pixels of different colors and not the color you had in mind.

The *TWinControl* Class (VCL)

In Windows, most elements of the user interface are windows. From a user standpoint, a window is a portion of the screen surrounded by a border, having a caption and usually a system menu. But technically speaking, a window is an entry in an internal system table, often corresponding to an element visible on the screen that has some associated code. Most of these windows have the role of controls; others are temporarily created by the system (for example, to show a pull-down menu). Still other windows are created by the application but remain hidden from the user and are used only as a way to receive a message (for example, nonblocking sockets use windows to communicate with the system).

The common denominator of all windows is that they are known by the Windows system and refer to a function for their behavior; each time something happens in the system, a notification message is sent to the proper window, which responds by executing some code. Each window of the system, in fact, has an associated function (generally called its *window procedure*), which handles the various messages the window is interested in.

In Delphi, any TWinControl class can override the WndProc method or define a new value for the WindowProc property. Interesting Windows messages, however, can be better tracked by providing specific message handlers. Even better, VCL converts these lower-level messages into events. In short, Delphi allows us to work at a high level, making application development easier, but still allows us to go low-level when this is required.

Notice also that creating a WinControl doesn't automatically create its corresponding Window handle. Delphi, in fact, uses a lazy initialization technique, so that the low control is only created when this is required, generally as soon as a method accesses the `Handle` property. The get method for this property the first time calls `HandleNeeded`, which eventually calls `CreateHandle`… and so on reaching `CreateWnd`, `CreateParams`, and `CreateWindowHandle` (the sequence is rather complex, and I don't think it is necessary to know it in detail). At the opposite end, you can keep an existing (perhaps invisible) control in memory but destroy its window handle, to save system resources.

The *TWidgetControl* Class (CLX)

In CLX, every `TWidgetControl` has an internal Qt object, referenced using the `Handle` property. This property has the same name as the corresponding Windows property, but it is totally different behind the scenes.

The Qt object is generally owned by the `TWidgetControl`, which automatically frees the object when it is destroyed. The class also uses delayed construction, as you can see in the `InitWidget` method, similar to `CreateWindow`. However it is also possible to create a widget around an existing Qt object: in this case, the widget won't own the Qt object and won't destroy it. The behavior is indicated by the `OwnHandle` property.

Actually each VisualCLX component has two associated C++ objects, the Qt `Handle` and the Qt `Hook`, which is the object receiving the system events. With the current Qt design, this has to be a C++ object, which acts as an intermediary to the event handlers of the Object Pascal control. The `HookEvents` method associates the hook object to the CLX control.

Differently from Windows, Qt defines two different types of events:

- *Events* are the translation of input or system events (such as key press, mouse move, and paint).
- *Signals* are internal component events (corresponding to VCL internal or abstract operations, such as `OnClick` and `OnChange`)

NOTE In CLX there is a seldom-used `EventHandler` method, which corresponds more or less to the `WndProc` method of VCL.

Opening the Component Tool Box

So you want to write a Delphi application. You open a new Delphi project and find yourself faced with a large number of components. The problem is that for every operation, there are multiple alternatives. For example, you can show a list of values using a list box, a combo box, a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which should you use? That's difficult to say. There are many considerations, depending on what you want your application to do. For this reason, I've provided a highly condensed summary of alternative options for a few common tasks.

NOTE For some of the controls described in the following sections, Delphi also includes a data-aware version, usually indicated by the *DB* prefix. As you'll see in Chapter 13, "Delphi's Database Architecture," the DB version of a control typically serves a role similar to that of its "standard" equivalent; but the properties and the ways you use it are often quite different. For example, in an Edit control you use the `Text` property, while in a DBEdit component you access the `Value` of the related field object.

The Text Input Components

Although a form or component can handle keyboard input directly, using the `OnKeyPress` event, this isn't a common operation. Windows provides ready-to-use controls you can use to get string input and even build a simple text editor. Delphi has several slightly different components in this area.

The Edit Component

The Edit component allows the user to enter a single line of text. You can also display a single line of text with a Label or a StaticText control, but these components are generally used only for fixed text or program-generated output, not for input. In CLX, there is also a native LCD digit control you can use to display numbers.

The Edit component uses the `Text` property, whereas many other controls use the `Caption` property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the `OnKeyPress` event of the edit box. For example, we can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's not, we change the value of the key to the null character (`#0`), so that it won't be processed by the edit control and will produce a warning beep:

```
procedure TForm1.Edit1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

NOTE A minor difference of CLX is that the Edit control has no Undo mechanism built in. Another is that the `PasswordChar` property is *replaced* by the `EchoMode` property. You don't determine the character to display, but whether to echo the entered text or display an asterisk instead.

The New LabeledEdit Control

Delphi 6 adds a very nice control, called LabeledEdit, which is an Edit control with a label attached to it. The Label appears as a property of the compound control, which inherits from `TCustomEdit`.

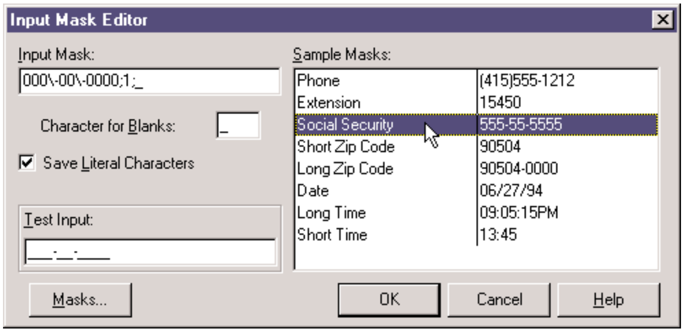
I have to say this component is very handy, because it allows you to reduce the number of components on your forms, move them around more easily, and have a more standard layout for labels, particularly when they are placed above the edit box. The `EditLabel` property is connected with the subcomponent, which has the usual properties and events. Two more properties, `LabelPosition` and `LabelSpacing`, allow you to configure the relative positions of the two controls.

NOTE This component has been added to the ExtCtrls unit to demonstrate the use of subcomponents in the Object Inspector, which is a new feature of Delphi 6. I'll discuss the development of these components in Chapter 11, "Creating Components." Notice also that this component, along with all of the other new Delphi 6 components, is not (yet) available on CLX and on the first release of Kylix. However, we can expect all non–Windows-specific additions to VCL, including subcomponents in general and the LabeledEdit control in particular, to be available in the next release of Kylix.

The MaskEdit Component

To customize the input of an edit box further, you can use the MaskEdit component, which has an `EditMask` property. This is a string indicating for each character whether it should be uppercase, lowercase, or a number, and other similar conditions. You can see the editor of the `EditMask` property in Figure 6.3.

FIGURE 6.3: The MaskEdit component's EditMask property editor



Input Mask: 000\-00\-0000;1;_

Character for Blanks: _

☑ Save Literal Characters

Test Input: ___-__-____

Sample Masks:

Phone	(415)555-1212
Extension	15450
Social Security	555-55-5555
Short Zip Code	90504
Long Zip Code	90504-0000
Date	06/27/94
Long Time	09:05:15PM
Short Time	13:45

TIP You can display any property's editor by selecting the property in the Object Inspector and clicking the ellipsis (…) button.

The Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to display the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated by semicolons).

TIP Clicking the Masks button of the Mask Editor lets you choose predefined input masks for different countries.

The Memo and RichEdit Components

Both of the controls discussed so far allow a single line of input. The Memo component, by contrast, can host several lines of text but (on the Win95/98 platforms) still retains the 16-bit Windows text limit (32 KB) and allows only a single font for the entire text. You can work on the text of the memo line by line (using the `Lines` string list) or access the entire text at once (using the `Text` property).

If you want to host a large amount of text or change fonts and paragraph alignments, in VCL you should use the RichEdit control, a Win32 common control based on the RTF document format. You can find an example of a complete editor based on the RichEdit component among the sample programs that ship with Delphi. (The example is named RichEdit, too.)

The RichEdit component has a `DefAttributes` property indicating the default styles and a `SelAttributes` property indicating the style of the current selection. These two properties are not of the `TFont` type, but they are compatible with fonts, so we can use the `Assign` method to copy the value, as in the following code fragment:

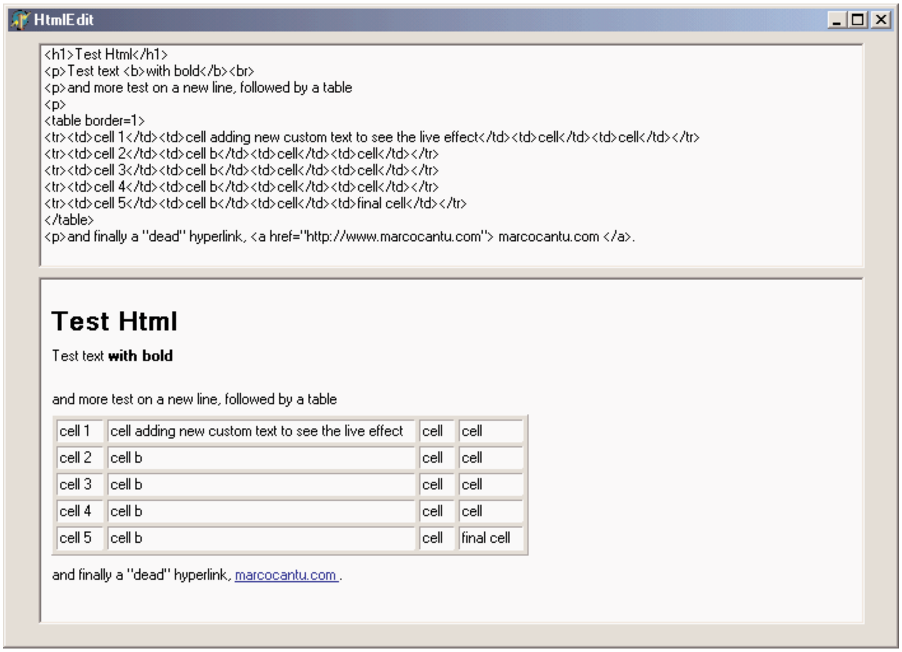
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign (RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  end;
end;
```

The TextViewer CLX Control

Among all of the common controls, CLX and Qt lack a RichEdit control. However, they provide a full-blown HTML viewer, which is very powerful for displaying formatted text but not for typing it. This HTML viewer is embedded in two different controls, the single-page TextViewer control or the TextBrowser control with active links.

As a simple demo, I've added a memo and a text viewer to a CLX form and connected them so that everything you type on the memo is immediately displayed in the viewer. I've called the example HtmlEdit not because this is a real HTML editor, but because this is the simplest way I know of to build an HTML preview inside a program. The form of the program is visible at run time in Figure 6.4, while typing some text inside a cell of the table.

FIGURE 6.4: The HtmlEdit example at run time: when you add new HTML text to the memo, you get an immediate preview.



```
<h1>Test Html</h1>
<p>Test text <b>with bold</b><br>
<p>and more test on a new line, followed by a table
<p>
<table border=1>
<tr><td>cell 1</td><td>cell adding new custom text to see the live effect</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 2</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 3</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 4</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 5</td><td>cell b</td><td>cell</td><td>final cell</td></tr>
</table>
<p>and finally a "dead" hyperlink, <a href="http://www.marcocantu.com"> marcocantu.com </a>.
```

Test Html

Test text **with bold**

and more test on a new line, followed by a table

cell 1	cell adding new custom text to see the live effect	cell	cell
cell 2	cell b	cell	cell
cell 3	cell b	cell	cell
cell 4	cell b	cell	cell
cell 5	cell b	cell	final cell

and finally a "dead" hyperlink, marcocantu.com .

TIP I originally built this example with Kylix on Linux. To port it to Windows and Delphi 6, all I had to do was to copy the files and recompile.

Selecting Options

There are two standard Windows controls that allow the user to choose different options, as well as controls for grouping sets of options.

The CheckBox and RadioButton Components

The first standard option-selecting control is the *check box*, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the `AllowGrayed` property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks the check box.

The second type of control is the *radio button*, which corresponds to an exclusive selection. Two radio buttons on the same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected (as programmer, you are responsible for selecting one of the radio buttons at design time).

The GroupBox Components

To host several groups of radio buttons, you can use a GroupBox control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the GroupBox component on a form and then add the radio buttons to the group box.

You can handle the radio buttons individually, but it's easier to navigate through the array of controls owned by the group box, as discussed in the previous chapter. Here is a small code excerpt used to get the text of the selected radio button of a group:

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := (GroupBox1.Controls[I] as TRadioButton).Caption;
```

The RadioGroup Component

Delphi has a similar component that can be used specifically for radio buttons: the RadioGroup component. A RadioGroup is a group box with some radio button *clones* painted inside it. The term *clone* in this context refers to the fact that the RadioGroup component is a single control, a single window, with elements similar to radio buttons painted on its surface.

Using the radio group is generally easier than using the group box, since the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Technically, a RadioGroup uses fewer resources and less memory, and it should be faster to create and paint. Also, the RadioGroup component can automatically align its radio buttons in one or more columns (as indicated by the `Columns` property), and you can easily add new choices at run time, by adding strings to the `Items` string list. By contrast, adding new radio buttons to a group box would be quite complex.

Lists

When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six, to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that display lists of items and allow the selection of one of them.

The ListBox Component

The selection of an item in a list box uses the `Items` and `ItemIndex` properties as in the code shown above for the RadioGroup control. If you need access to the text of selected list box items often, you can write a small wrapper function like this:

```
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := '';
end;
```

Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the `MultiSelect` property. There are two kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case, a user selects multiple items simply by clicking them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items, respectively. This second choice is determined by the `ExtendedSelect` property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the `SelCount` property, and it can determine which items are selected by examining the `Selected` array. This array of Boolean values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the `Selected` array as follows:

```
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```

In CLX the ListBox can be configured to use a fixed number of columns and rows, using the `Columns`, `Row`, `ColumnLayout` and `RowLayout` properties. Of these, the VCL ListBox has only the `Columns` property.

The ComboBox Component

List boxes take up a lot of screen space, and they offer a fixed selection—that is, a user can choose only among the items in the list box and cannot enter any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control, which combines an edit box and a drop-down list. The behavior of a ComboBox component changes a lot depending on the value of its `Style` property:

- The csDropDown style defines a typical combo box, which allows direct editing and displays a list box on request.
- The csDropDownList style defines a combo box that does not allow editing (but uses the keystrokes to select an item).
- The csSimple style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a ComboBox is easier than doing the same operation for a list box, since you can simply use the `Text` property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the Enter key. The following method first tests whether the user has pressed that key, by looking for the character with the numeric (ASCII) value of 13. It then tests to make sure the text of the combo box is not empty and is not already in the list—if its position in the list is less than zero. Here is the code:

```
procedure TForm1.ComboBox1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // if the user presses the Enter key
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

NOTE In CLX, the combo box can automatically add the text typed into the edit to the drop-down list, when the user presses the Enter key. Also, some events fire at different times than in VCL.

Delphi 6 includes two new events for the combo box. The `OnCloseUp` event corresponds to the closing of the drop-down list and complements the preexisting `OnDropDown` event. The `OnSelect` event fires only when the user selects something in the drop-down list, as opposed to typing in the edit portion.

Another very nice addition is the `AutoComplete` property. When it is set, the ComboBox component (and the ListBox, as well) automatically locates the string nearest to the one the user is entering, suggesting the final part of the text. The core of this feature, available also in CLX, is implemented in the `TCustomListBox.KeyPress` method.

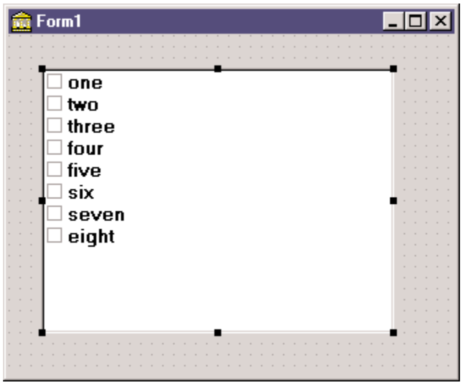
The CheckListBox Component

Another extension of the list box control is represented by the CheckListBox component, a list box with each item preceded by a check box (as you can see in Figure 6.5). A user can select a single item of the list, but can also click the check boxes to toggle their status. This makes the CheckListBox a very good component for multiple selections or for highlighting the status of a series of independent items (as in a series of check boxes).

To check the current status of each item, you can use the `Checked` and the `State` array properties (use the latter if the check boxes can be grayed). Delphi 5 introduced the `ItemEnabled` array property, which you can use to enable or disable each item of the list. We'll use the CheckListBox in the DragList example, later in this chapter.

FIGURE 6.5:

The user interface of the CheckListBox control, basically a list of check boxes



TIP

Most of the list-based controls share a common and important feature. Each item of the list has an associated 32-bit value, usually indicated by the TObject type. This value can be used as a tag for each list item, and it's very useful for storing additional information along with each item. This approach is connected to a specific feature of the native Windows list box control, which offers four bytes of extra storage for each list box item. We'll use this feature in the ODList example later on in this chapter.

New Combo Boxes: ComboBoxEx and ColorBox

The ComboBoxEx (where *ex* stands for extended) is the wrapper of a new Win32 common controls, which extends the traditional combo box by allowing images to appear next to the items in the list. You attach an image list to the combo, and then select an image index for each item to display. The effect of this change is that the simple Items string list is replaced by a more complex collection, the ItemsEx property.

The ColorBox control is a new version of the combo box specifically aimed at selecting colors. You can use its Style property for choosing which groups of colors you want to see in the list (standard color, extended colors, system colors, and so on).

The ListView and TreeView Components

If you want an even more sophisticated list, you can use the ListView common control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described at the beginning of the next chapter, "Advanced VCL Controls." Other alternatives for listing values are the TreeView common control, which shows items in a hierarchical output, and the StringGrid control, which shows multiple elements for each line. The string grid control is described in the "Graphics in Delphi" bonus chapter, available on the companion CD.

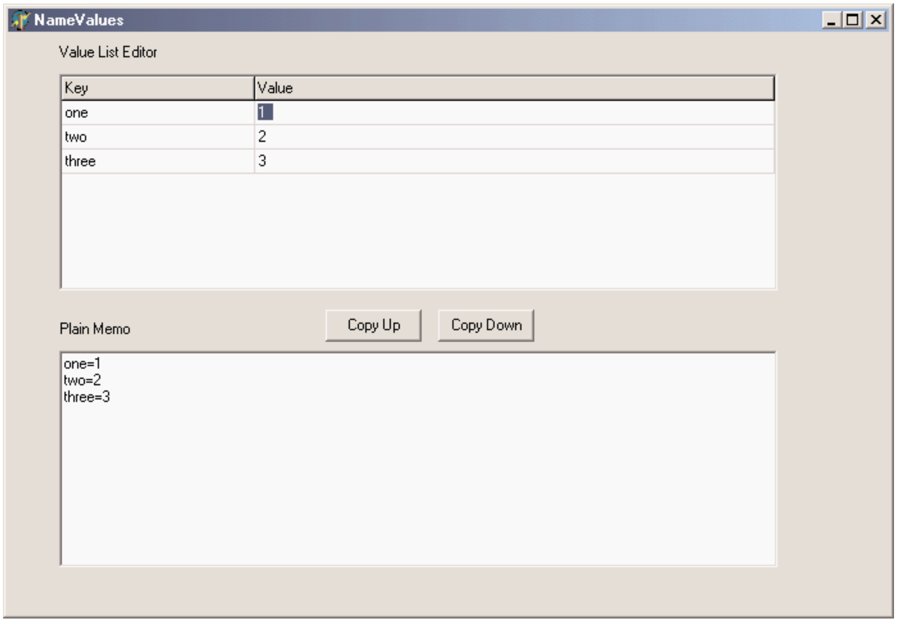
If you use the common controls in your application, users will already know how to interact with them, and they will regard the user interface of your program as up to date. TreeView and ListView are the two key components of Windows Explorer, and you can assume that many users will be familiar with them, even more than with the traditional Windows controls. CLX adds also an IconView control, which parallels part of the features of the VCL ListView.

The New ValueListEditor Component

Delphi applications often use the name/value structure natively offered by string lists, which I discussed in the last chapter. Delphi 6 introduces a version of the StringGrid component specifically geared towards this type of string lists. The ValueListEditor has two columns where you can display and let the user edit the contents of a string list with name/value pairs, as you can see in Figure 6.6. This string list is indicated in the Strings property of the control.

FIGURE 6.6:

The NameValues example has the new ValueListEditor component, which shows the name/value or key/value pairs of a string list, visible also in a plain memo.



Key	Value
one	1
two	2
three	3

```
one=1
two=2
three=3
```

The power of this control lies in the fact you can customize the editing options for each position of the grid or for each key value, using the run-time-only `ItemProps` array property. For each item, you can indicate:

- Whether it is read-only
- The maximum number of characters of the string
- An edit mask (eventually requested in the `OnGetEditMask` event)
- The items of a drop-down pick list (eventually requested in the `OnGetPickList` event)
- The display of a button for showing an editing dialog (in the `OnEditButtonClick` event)

Needless to say, this behavior resembles what is available generally for string grids and the DBGrid control, but also the behavior of the Object Inspector.

The `ItemProps` property has to be set up at run time, by creating an object of the `TItemProp` class and assigning it to an index or a key of the string list. To have a default editor for each line, you can assign the same item property object multiple times. In the example, this shared editor sets an edit mask for up to three numbers:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
```

```
begin
  SharedItemProp := TItemProp.Create (ValueListEditor1);
  SharedItemProp.EditMask := '999;0; ';

  Memo1.Lines := ValueListEditor1.Strings;
  for I := 0 to ValueListEditor1.Strings.Count - 1 do
    ValueListEditor1.ItemProps [I] := SharedItemProp;
end;
```

Similar code has to be repeated in case the number of lines changes—for example, by adding new elements in the memo and copying them up to the value list:

```
procedure TForm1.ValueListEditor1StringsChange(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ValueListEditor1.Strings.Count - 1 do
    if not Assigned (ValueListEditor1.ItemProps [I]) then
      ValueListEditor1.ItemProps [I] := SharedItemProp;
end;
```

NOTE Apparently reassigning the same editor twice causes some trouble, so I've assigned the editor only to the lines not having already one.

Another property, KeyOptions, allows you to let the user also edit the keys (the names), add new entries, delete existing ones, and allow for duplicated names in the first portion of the string. Oddly enough, you cannot add new keys unless you also activate the edit options, which makes it hard to let the user add extra entries while preserving the names of the basic ones.

Ranges

Finally, there are a few components you can use to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

The ScrollBar Component

The stand-alone ScrollBar control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border, a feature governed by a specific Windows style of the form window. By *resembling*, I mean that it is not technically a separate window of the ScrollBar component type. These "fake" scroll bars are usually controlled in Delphi using specific properties of the form and the other components hosting them.

The TrackBar and ProgressBar Components

Direct use of the ScrollBar component is quite rare, especially with the TrackBar component introduced with Windows 95, which is used to let a user select a value in a range. Among Win32 common controls is the companion ProgressBar control, which allows the program to output a value in a range, showing the progress of a lengthy operation.

The UpDown Component

Another related control is the UpDown component, which is usually connected to an edit box so that the user can either type a number in it or increase and decrease the number using the two small arrow buttons. To connect the two controls, you set the `Associate` property of the UpDown component. Nothing prevents you from using the UpDown component as a standalone control, displaying the current value in a label or in some other way.

NOTE In CLX there is no UpDown control, but a SpinEdit that bundles an Edit with the UpDown in a single control.

The PageScroller Component

The Win32 PageScroller control is a container allowing you to scroll the internal control. For example, if you place a toolbar in the page scroller and the toolbar is larger than the available area, the PageScroller will display two small arrows on the side. Clicking these arrows will scroll the internal area. This component can be used as a scrollbar, but it also partially replaces the ScrollBox control.

The ScrollBox Component

The ScrollBox control represents a region of a form that can scroll independently from the rest of the surface. For this reason, the ScrollBox has two scrollbars used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many Windows applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scrollbars of the form, in fact, you might allow the user to move the toolbar or status bar out of view, a very odd situation.

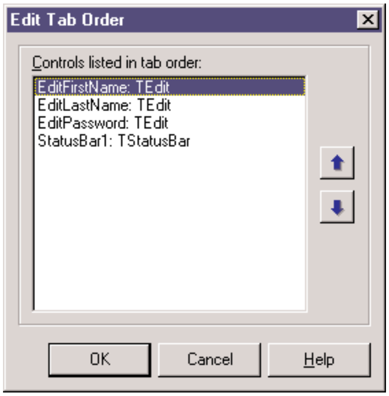
Handling the Input Focus

Using the `TabStop` and `TabOrder` properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key.

Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, as shown in Figure 6.7.

FIGURE 6.7:

The Edit Tab Order dialog box



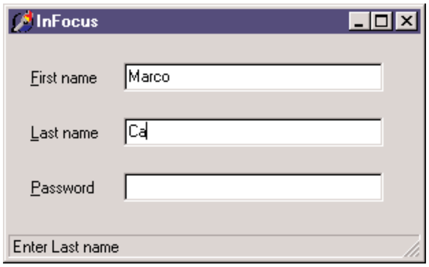
Controls listed in tab order:

EditFirstName: TEdit
EditLastName: TEdit
EditPassword: TEdit
StatusBar1: TStatusBar

Besides these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the InFocus example, which creates a fairly typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in Figure 6.8. At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.

FIGURE 6.8:

The InFocus example at run time



First name: Marco
Last name: Ca
Password:
Enter Last name

For the output of the status information, I've used the StatusBar component, with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the `&` character in the labels, indicating a shortcut key, and the connection of these labels with corresponding edit boxes (using the FocusControl property):

```
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
  object Label1: TLabel
    Caption = '&First name'
    FocusControl = EditFirstName
  end
  object EditFirstName: TEdit
    OnEnter = GlobalEnter
    OnExit = EditFirstNameExit
  end
  object Label2: TLabel
    Caption = '&Last name'
    FocusControl = EditLastName
  end
  object EditLastName: TEdit
    OnEnter = GlobalEnter
  end
  object Label3: TLabel
    Caption = '&Password'
    FocusControl = EditPassword
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    OnEnter = GlobalEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```

The program is very simple and does only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' OnEnter event, possibly using a single generic event handler to avoid repetitive code. In the example, instead of storing some extra information for each edit box, I've checked each control of the form to determine which label is connected to the current edit box (indicated by the Sender parameter):

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    // if the control is a label
    if (Controls [I] is TLabel) and
      // and the label is connected to the current edit box
      (TLabel(Controls[I]).FocusControl = Sender) then
    // copy the text, leaving off the initial & character
    StatusBar1.SimpleText := 'Enter ' +
      Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;
```

The second event handler of the form relates to the `OnExit` event of the first edit box. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```
procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
  if EditFirstName.Text = '' then
  begin
    // don't let the user get out
    EditFirstName.SetFocus;
    MessageDlg ('First name is required', mtError, [mbOK], 0);
  end
  else if EditFirstName.Text = 'Admin' then
  begin
    // fill the second edit and jump to the third
    EditLastName.Text := 'Admin';
    EditPassword.SetFocus;
  end;
end;
```

TIP The CLX version of this example has exactly the same code and is available as the QInFocus program. The same happens for most of the other examples of this chapter. Notice that some of the examples are quite complex, but I rarely had to touch the code at all.

Working with Menus

Working with menus and menu items is generally quite simple. This section offers only some very brief notes and a few more advanced examples. The first thing to keep in mind about menu items is that they can serve different purposes:

Commands are menu items used to execute an action.

State-setters are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active. In Delphi 6 you can automatically obtain this behavior using the handy AutoCheck property.

Radio items have a round check mark and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, simply set the RadioItem property to True and set the GroupIndex property for the alternative menu items to the same value.

Dialog menu items cause a dialog box to appear and are usually indicated by an ellipsis (three dots) after the text.

As you enter new elements in the Menu Designer, Delphi creates a new component for each menu item and lists it in the Object Inspector (although nothing is added to the form). To name each component, Delphi uses the caption you enter and appends a number (so that *Open* becomes Open1). Because Delphi removes spaces and other special characters in the caption when it creates the name, and the menu item separators are set up using a hyphen as caption, these items would have an empty name. For this reason Delphi adds the letter *N* to the name, appending the number and generating items called N1, N2, and so on.

WARNING Do not use the Break property, which is used to lay out a pull-down menu on multiple columns. The mbMenuBarBreak value indicates that this item will be displayed in a second or subsequent line; the mbMenuBreak value that this item will be added to a second or subsequent column of the pull-down.

Accelerator Keys

Since Delphi 5, you don't need to enter the & character in the Caption of a menu item; it provides an automatic accelerator key if you omit one. Delphi's automatic accelerator-key system can also figure out if you have entered conflicting accelerator keys and fix them on-the-fly. This doesn't mean you should stop adding custom accelerator keys with the & character, because the automatic system simply uses the first available letter, and it doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This feature is controlled by the AutoHotkeys property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to maAutomatic, while in the pull-downs and menu items it defaults to maParent, so that the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of maAutomatic or maManual.

The engine behind this system is the RethinkHotkeys method of the TMenuItem class, and the companion InternalRethinkHotkeys. There is also a RethinkLines method, which checks whether a pull-down has two consecutive separators or begins or ends with a separator. In all these cases, the separator is automatically removed.

One of the reasons Delphi includes this feature is the Integrated Translation Environment (ITE). When you need to translate the menu of an application, it is convenient if you don't have to deal with the accelerator keys, or at least if you don't have to worry about whether two items on the same menu conflict. Having a system that can automatically resolve similar problems is definitely an advantage. Another motivation was Delphi's IDE itself. With all the dynamically loaded packages that install menu items in the IDE main menu or in pop-up menus, and with different packages loaded in different versions of the product, it's next to impossible to get nonconflicting accelerator-key selections in each menu. That is why this mechanism isn't a wizard that does static analysis of your menus at design time; it was created to deal with the real problem of managing menus created dynamically at run time.

WARNING This feature is certainly very handy, but because it is active by default, it can break existing code. I had to modify two of this chapter's program examples, between the Delphi 4 and Delphi 5 edition of the book, just to avoid run-time errors caused by this change. The problem is that I use the caption in the code, and the extra *&* broke my code. The change was quite simple, though: All I had to do was to set the `AutoHotkeys` property of the main menu component to maManual.

Pop-Up Menus and the *OnContextPopup* Event

Besides the MainMenu component, you can use the similar PopupMenu component. This is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property.

However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, in this code fragment a label:

```
procedure TForm1.Label3MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
  begin
    ScreenPoint := Label3.ClientToScreen (Point (X, Y));
    PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
  end;
end;
```

An alternative approach is the use of the OnContextMenu event. This event, introduced in Delphi 5, fires when a user right-clicks a component—exactly what we've traced above with the test if Button = mbRight. The advantage is that the same event is also fired in response to a Shift+F10 key combination, as well as by any other user-input methods defined by Windows Accessibility options or hardware (including the shortcut-menu key of some Windows-compatible keyboards). We can use this event to fire a pop-up menu with little code:

```
procedure TFormPopup.Label1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now), 0, False, True, nil, 0, ''));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This is not particularly useful, but I've done it to highlight that if you need to display a plain pop-up menu, you can easily use the PopupMenu property of the control in question or one of its parent controls. Handling the OnContextMenu event makes sense only when you want to do some extra processing.

The Handled parameter is preinitialized to False, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set Handled to True and the system will stop processing the message. Setting Handled to True should be fairly rare, as you'll generally handle the OnContextPopup to dynamically create or customize the pop-up menu, but then you can let the default handler actually show the menu.

The handler of an OnContextPopup event isn't limited to displaying a pop-up menu. It can do any other operation, such as directly display a dialog box. Here is an example of a right-click operation used to change the color of the control:

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
```

```
    ColorDialog1.Color := Label2.Color;
    if ColorDialog1.Execute then
      Label2.Color := ColorDialog1.Color;
    Handled := True;
  end;
```

All the code snippets of this section are available in the simple CustPop example for VCL and QCustPop for CLX, on the book's companion CD.

Creating Menu Items Dynamically

Besides defining the structure of a menu with the Menu Designer and modifying the status of the items using the Checked, Visible, and Caption properties, you can create an entire menu or portions of one at run time. This makes sense, for example, when you have many repetitive items, or when the menu items depend on some system configuration or user permissions.

The basic idea is that each object of the TMenuItem class—which Delphi uses for both menu items and pull-down menus—contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each submenu has a list of submenus, each with its own list of submenus, and so on. The properties you can use to explore the structure of an existing menu are Items, which contains the actual list of menu items, and Count, which contains the number of subitems. Adding new menu items or entire pull-down menus to an existing menu is fairly easy, particularly if you can write a single event handler for all of them.

This is demonstrated by the DynaMenu example (and its QDynaMenu counterpart), which also illustrates the use of menu check marks, radio items, and many other features of menus that aren't described in detail in the text. As soon as you start this program, it creates a new pull-down with menu items used to change the font size of a big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you.

The new pull-down menu should be inserted in the Items property of the MainMenu1 component. You can calculate the position by asking the MainMenu component for the previous pull-down menu:

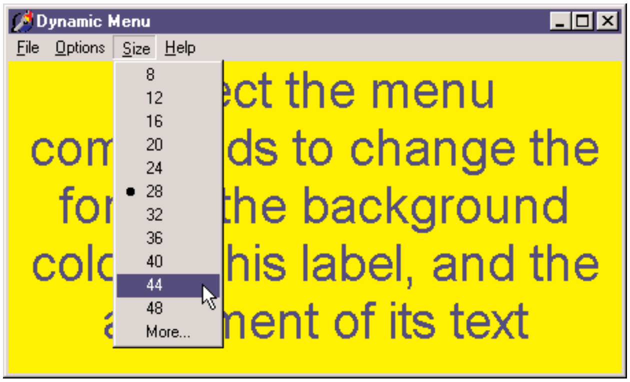
```
procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pull-down menu
  PullDown := TMenuItem.Create (Self);
  PullDown.AutoHotkeys := maManual;
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
```

```
    Position := MainMenu1.Items.IndexOf (Options1);
    MainMenu1.Items.Insert (Position + 1, PullDown);
    // create menu items for various sizes
    I := 8;
    while I <= 48 do
    begin
      // create the new item
      Item := TMenuItem.Create (Self);
      Item.Caption := IntToStr (I);
      // make it a radio item
      Item.GroupIndex := 1;
      Item.RadioItem := True;
      // handle click and insert
      Item.OnClick := SizeItemClick;
      PullDown.Insert (PullDown.Count, Item);
      I := I + 4;
    end;
    // add extra item at the end
    Item := TMenuItem.Create (Self);
    Item.Caption := 'More...';
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle it by showing the font selection dialog
    Item.OnClick := Font1Click;
    PullDown.Insert (PullDown.Count, Item);
  end;
```

As you can see in the preceding code, the menu items are created in a `while` loop, setting the radio item style and calling the `Insert` method with the number of items as a parameter to add each item at the end of the pull-down. At the end, the program adds one extra item, which is used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method (also connected to a specific menu item), which displays the font selection dialog box. You can see the dynamic menu in Figure 6.9.

FIGURE 6.9:

The Size pull-down menu of the DynaMenu example is created at run time, along with all of its menu items.



ect the menu
com ds to change the
for the background
colc his label, and the
a ment of its text

WARNING Because the program uses the Caption of the new items dynamically, we should either disable the AutoHotkeys property of the main menu component, or disable this feature for the pull-down menu we are going to add (and thus automatically disable it for the menu items). This is what I've done in the code above by setting the AutoHotkeys property of the dynamically created pull-down component to maManual. An alternative approach is to let the menu display the automatic captions and then call the new StripHotkeys function before converting the caption to a number. There is also a new GetHotkey function, which returns the *active* character of the caption.

The handler for the OnClick event of these dynamically created menu items uses the caption of the Sender menu item to set the size of the font:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
end;
```

This code doesn't set the proper radio-item mark next to the selected item, because the user can select a new size also by changing the font. The proper radio item is checked in the OnClick event handler of the entire pull-down menu, which is connected just after the pull-down is created and activated just before showing the pull-down. The code scans the items of the pull-down menu (the Sender object) and checks whether the caption matches the current Size of the font. If no match is found, the program checks the last menu item, to indicate that a different size is active:

```
procedure TFormColorText.SizeClick (Sender: TObject);
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) = Label1.Font.Size then
      begin
        Items [I].Checked := True;
        Found := True;
        System.Break; // skip the rest of the loop
      end;
    if not Found then
      Items [Count - 1].Checked := True;
  end;
end;
```

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the DynaMenu and QDynaMenu examples. As an alternative, you can also use some global functions available in the Menus unit: `NewMenu`, `NewPopupMenu`, `NewSubMenu`, `NewItem`, and `NewLine`.

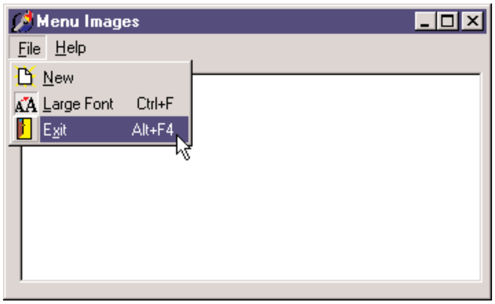
Using Menu Images

In Delphi it is very easy to improve a program's user interface by adding images to menu items. This is becoming common in Windows applications, and it's very nice that Borland has added all the required support, making the development of graphical menu items trivial.

All you have to do is add an image list control to the form, add a series of bitmaps to the image list, connect the image list to the menu using its `Images` property, and set the proper `ImageIndex` property for the menu items. You can see the effect of these simple operations in Figure 6.10. (You can also associate a bitmap with the menu item directly, using the `Bitmap` property.)

FIGURE 6.10:

The simple graphical menu of the MenuImg example



TIP The definition of images for menus is quite flexible, as it allows you to associate an image list with any specific pull-down menu (and even a specific menu item) using the `SubMenuImages` property. Having a specific and smaller image list for each pull-down menu, instead of one single huge image list for the entire menu, allows for more run-time customization of an application.

To create the image list, double-click the component, activating the corresponding editor (shown in Figure 6.11), then import existing bitmap or icon files. You can actually prepare a single large bitmap and let the image editor divide it according to the `Height` and `Width` properties of the ImageList component, which refer to the size of the individual bitmaps in the list.

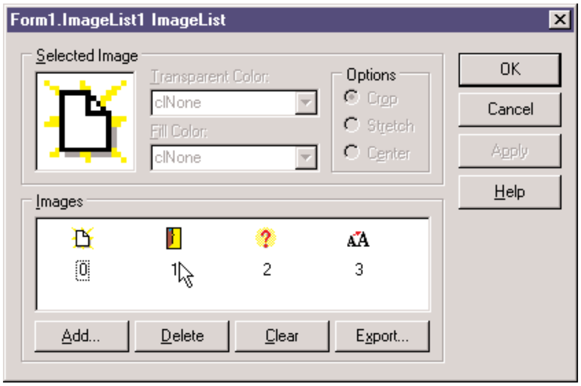


FIGURE 6.11:

The Image List editor, with the bitmaps of the MenuImg example

TIP

As an alternative, you can use the series of images that ship with Delphi and are stored by default in the `\Program Files\Common Files\Borland Shared\Images\Buttons` directory. Each bitmap contains both an "enabled" and a "disabled" image. As you import them, the Image List editor will ask you whether to split them in two, a suggestion you should accept. This operation adds to the image list a normal image and a disabled one, which is not generally used (as it can be built automatically when needed). For this reason I generally delete the disabled part of the bitmap from the image list.

The program's code is very simple. The only element I want to emphasize is that if you set the `Checked` property of a menu item with an image instead of displaying a check mark, the item paints its image as "sunken" or "recessed." You can see this in the Large Font menu of the MenuImg example in Figure 6.10. Here is the code for that menu item selection:

```
procedure TForm1.LargeFont1Click(Sender: TObject);
begin
  if Memo1.Font.Size = 8 then
    Memo1.Font.Size := 12
  else
    Memo1.Font.Size := 8;
  // changes the image style near the item
  LargeFont1.Checked := not LargeFont1.Checked;
end;
```

WARNING

To make the CLX version of the program, QMenuImg, display the bitmaps properly, I had to reimport them. Simply converting the Image List component data didn't work.

Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and "quick-and-dirty" applications. Adding a single menu item to the system menu is straightforward:

```
AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE), MF_STRING, idSysAbout, '&About...');
```

This code fragment (extracted from the `OnCreate` event handler of the SysMenu example) adds a separator and a new item to the system menu item. The `GetSystemMenu` API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The `AppendMenu` API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I've defined this identifier as:

```
const idSysAbout = 100;
```

Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the `wm_Command` Windows message. This is handled internally by Delphi, which activates the `OnClick` event of the corresponding menu item component. The selection of system menu commands, instead, generates a `wm_SysCommand` message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.

We can intercept this command and check to see whether the command identifier (passed in the `CmdType` field of the `TWmSysCommand` parameter) of the menu item is idSysAbout. Since there isn't a corresponding event in Delphi, we have to define a new message-response method for the form class:

```
public
  procedure WMSysCommand (var Msg: TMessage);
    message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a MainMenu component to the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the Menu property of the form, removing the menu bar.

Now we can add some code to the SysMenu example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is clicked. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

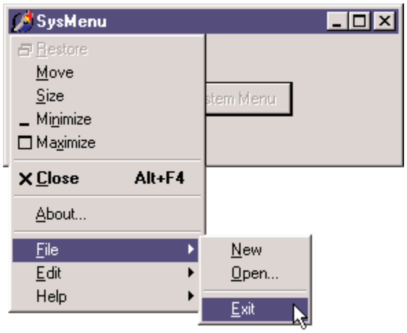
```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (Self.Handle, FALSE),
        mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

TIP This code uses the expression Self.Handle to access the handle of the form. This is required because we are currently working on the MainMenu1 component, as specified by the with statement.

The menu flag used in this case, mf_Popup, indicates that we are adding a pull-down menu. In this function call, the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example, we passed the identifier of the menu, instead). Since we are adding to the system menu items with submenus, the final structure of the system menu will have two levels, as you can see in Figure 6.12.

FIGURE 6.12:

The second-level system menu items of the SysMenu example are the result of copying a complete main menu to the system menu.



WARNING The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms to mean different things. Pop-up menus are shortcut menus, and pull-down menus are the secondary menus of the menu bar. Apparently, Microsoft uses the terms in this way because the two elements are implemented with the same kind of internal windows; the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course, you can check for each menu item in the WMSysCommand method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the OnClick event of each item, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a FindItem method.

When (and if) we have found a main menu item that corresponds to the item selected in the system menu, we can call its Click method (which invokes the OnClick handler). Here is the code I've added to the WMSysCommand method:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
  if Item <> nil then
    Item.Click;
```

In this code, the CmdType field of the message structure that is passed to the WMSysCommand procedure holds the command of the menu item being called.

You can also use a simple if or case statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as sc_Close, sc_Minimize, sc_Maximize, and so on. For more information, you can see the description of the wm_SysCommand message in the Windows API Help file.

This application works but has one glitch. If you click the right mouse button over the Taskbar icon representing the application, you get a plain system menu (actually different from the default one). The reason is that this system menu belongs to a different window, the window of the Application global object. I'll discuss the Application object, and update this example to make it work with the Taskbar button, in Chapter 9, "Working with Forms."

NOTE Because this program uses low-level Windows features (API calls and messages), it is not possible to compile it with CLX, so there is no Qt version of this example.

Owner-Draw Controls and Styles

Let's return briefly to menu graphics. Besides using an ImageList to add glyphs to the menu items, you can turn a menu into a completely graphical element, using the owner-draw technique. The same technique also works for other controls, such as list boxes. In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically, these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In VCL, the situation is slightly more complex. The components can take care of painting themselves in this case (as in the `TBitBtn` class for bitmap buttons) and possibly activate corresponding events. The system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers.

In CLX, some of the controls, such as ListBoxes and ComboBoxes, surface events very similar to Windows owner-draw, but menus lack them. The native approach of Qt is to use styles to determine the graphical behavior of all of the controls in the system, of a specific application, or of a given control. I'll introduce styles shortly, later in this section.

NOTE Most of the Win32 common controls have support for the owner-draw technique, generally called *custom drawing*. You can fully customize the appearance of a ListView, TreeView, TabControl, PageControl, HeaderControl, StatusBar, and ToolBar. The ToolBar, ListView, and TreeView controls also support *advanced* custom drawing, a more fine-tuned drawing capability introduced by Microsoft in the latest versions of the Win32 common controls library. The downside to owner-draw is that when the Windows user interface style changes in the future (and it always does), your owner-draw controls that fit in perfectly with the current user interface styles will look outdated and out of place. Since you are creating a custom user interface, you'll need to keep it updated yourself. By contrast, if you use the standard output of the controls, your applications will automatically adapt to a new version of such controls.

Owner-Draw Menu Items

VCL makes the development of graphical menu items quite simple compared to the traditional approach of the Windows API: You set the `OwnerDraw` property of a menu item component to True and handle its `OnMeasureItem` and `OnDrawItem` events. This same feature is not available on CLX.

In the `OnMeasureItem` event, you can determine the size of the menu items. This event handler is activated once for each menu item when the pull-down menu is displayed and has two reference parameters you can set:

```
procedure ColorMeasureItem (Sender: TObject; ACanvas: TCanvas;
  var Width, Height: Integer);
```

The other parameter, ACanvas, is typically used to determine the height of the current font.

In the OnDrawItem event, you paint the actual image. This event handler is activated every time the item has to be repainted. This happens when Windows first displays the items and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on. Luckily enough, the Delphi event passes to the handler the Canvas where it should paint, the output rectangle, and the status of the item (selected or not):

```
procedure ColorDrawItem(Sender: TObject; ACanvas: TCanvas; ARect: TRect;
  Selected: Boolean);
```

In the ODMenu example, I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've set the OwnerDraw property of the menu and written handlers for some of the menu items. To write a single handler for each event of the three color-related menu items, I've set their Tag property to the value of the actual color in the OnCreate event handler of the form. This makes the handler of the actual OnClick event of the items quite straightforward:

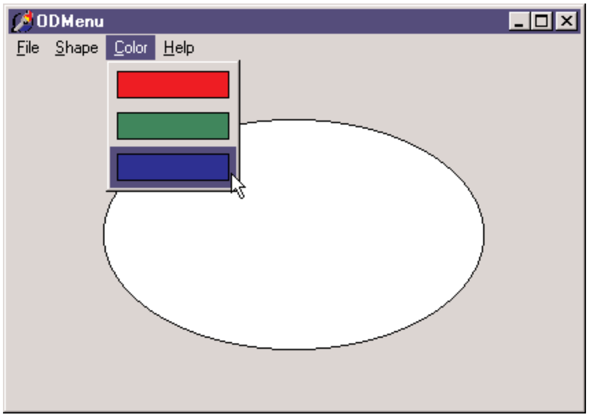
```
procedure TForm1.ColorClick(Sender: TObject);
begin
  ShapeDemo.Brush.Color := (Sender as TComponent).Tag
end;
```

The handler of the OnMeasureItem event doesn't depend on the actual items, but uses fixed values (different from the handler of the other pull-down). The most important portion of the code is in the handlers of the OnDrawItem events. For the color, we use the value of the tag to paint a rectangle of the given color, as you can see in Figure 6.13. Before doing this, however, we have to fill the background of the menu items (the rectangular area passed as a parameter) with the standard color for the menu (clMenu) or the selected menu items (clHighlight):

```
procedure TForm1.ColorDrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // show the color
  ACanvas.Brush.Color := (Sender as TComponent).Tag;
  InflateRect (ARect, -5, -5);
  ACanvas.Rectangle (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

FIGURE 6.13:

The owner-draw menu of the ODMenu example



The three handlers for this event of the Shape pull-down menu items are all different, although they use similar code:

```
procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // draw the ellipse
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

NOTE To accommodate the increasing number of states in the Windows 2000 user interface style, since version 5, Delphi has included the `OnAdvancedDrawItem` event for menus.

A ListBox of Colors

As we have just seen for menus, list boxes have an owner-draw capability, which means a program can paint the items of a list box. The same support is provided for combo boxes and is also available on CLX. To create an owner-draw list box, we set its `Style` property to lbOwnerDrawFixed or lbOwnerDrawVariable. The first value indicates that we are going to set the height of the items of the list box by specifying the `ItemHeight` property and that this will be the height of each and every item. The second owner-draw style indicates a list box with items of different heights; in this case, the component will trigger the `OnMeasureItem` event for each item, to ask the program for their heights.

In the ODList example (and its QODList version), I'll stick with the first, simpler, approach. The example stores color information along with the items of the list box and then draws the items in colors (instead of using a single color for the whole list).

The DFM or XFM file of every form, including this one, has a `TextHeight` attribute, which indicates the number of pixels required to display text. This is the value we should use for the `ItemHeight` property of the list box. An alternative solution is to compute this value at run time, so that if we later change the font at design time, we don't have to remember to set the height of the items accordingly.

NOTE I've just described `TextHeight` as an *attribute* of the form, not a property. And in fact it isn't a property but a local value of the form. If it is not a property, you might ask, how does Delphi save it in the `DFM` file? Well, the answer is that Delphi's streaming mechanism is based on properties plus special *property clones* created by the `DefineProperties` method.

Since `TextHeight` is *not* a property, although it is listed in the form description, we cannot access it directly. Studying the VCL source code, I found that this value is computed by calling a private method of the form: `GetTextHeight`. Since it is private, we cannot call this function. What we can do is duplicate its code (which is actually quite simple) in the `FormCreate` method of the form, after selecting the font of the list box:

```
Canvas.Font := ListBox1.Font;
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

The next thing we have to do is add some items to the list box. Since this is a list box of colors, we want to add color names to the `Items` of the list box and the corresponding color values to the `Objects` data storage related to each item of the list. Instead of adding the two values separately, I've written a procedure to add new items to the list:

```
procedure TODListForm.AddColors (Colors: array of TColor);
var
  I: Integer;
begin
  for I := Low (Colors) to High (Colors) do
    ListBox1.Items.AddObject (ColorToString (Colors[I]), TObject(Colors[I]));
end;
```

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. For each item passed as a parameter, we add the name of the color to the list, and we add its value to the related data, by calling the `AddObject` method. To obtain the string corresponding to the color, we call the Delphi `ColorToString` function. This returns a string containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the TObject data type (a four-byte reference), as required by the AddObject method.

TIP Besides ColorToString, which converts a color value into the corresponding string with the identifier or the hexadecimal value, there is also a Delphi function to convert a properly formatted string into a color, StringToColor.

In the ODList example, this method is called in the OnCreate event handler of the form (after previously setting the height of the items):

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime, clPurple,
  clGray, RGB (213, 23, 123), RGB (0, 0, 0), clAqua, clNavy, clOlive, clTeal]);
```

To compile the CLX version of this code, I've added to it the RGB function described earlier in the section "Colors." The code used to draw the items is not particularly complex. We simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  with Control as TListbox do
  begin
    // erase
    Canvas.FillRect(Rect);
    // draw item
    Canvas.Font.Color := TColor (Items.Objects [Index]);
    Canvas.TextOut(Rect.Left, Rect.Top, Listbox1.Items[Index]);
   end;
end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on our part. You can see an example of the output of this program at startup in Figure 6.14.

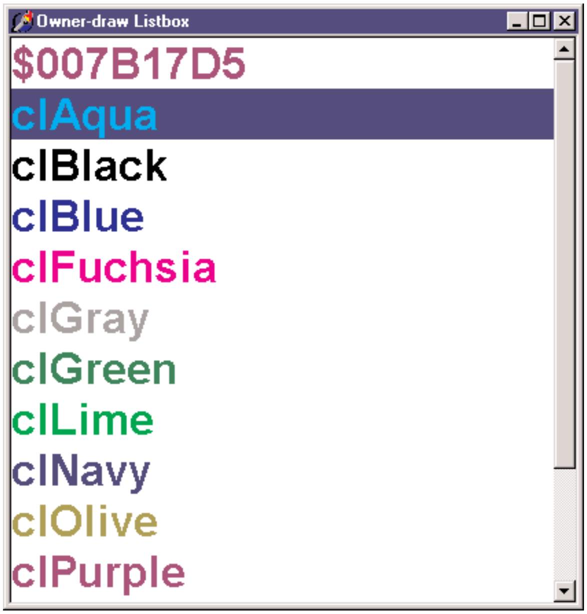
The example also allows you to add new items, by double-clicking the list box:

```
procedure TODListForm.ListBox1DblClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    AddColors ([ColorDialog1.Color]);
end;
```

If you try using this capability, you'll notice that some colors you add are turned into color names (one of the Delphi color constants) while others are converted into hexadecimal numbers.

FIGURE 6.14:

The output of the ODList example, with a colored owner-draw list box



CLX Styles

In Windows, the system has full control of the user interface of the controls, unless the program takes over using owner-draw or other advanced techniques. In Qt (and in Linux in general), the user chooses the user interface style of the controls. A system will generally offer a few basic styles, such as the Windows look-and-feel, the Motif one, and others. A user can add also install new styles in the system and make them available to applications.

NOTE The styles I'm discussing here refer to the user interface of the controls, not of the forms and their borders. This is generally configurable on Linux systems but is technically a separate element of the user interface.

Because this technique is embedded in Qt, it is also available on the Windows version of the library, and CLX makes it available to Delphi developers. The `Application` global object of CLX has a `Style` property, which can be used to set a custom style or a default one, indicated by the `DefaultStyle` subproperty. For example, you can select a Motif look-and-feel with this code:

```
Application.Style.DefaultStyle := dsMotif;
```

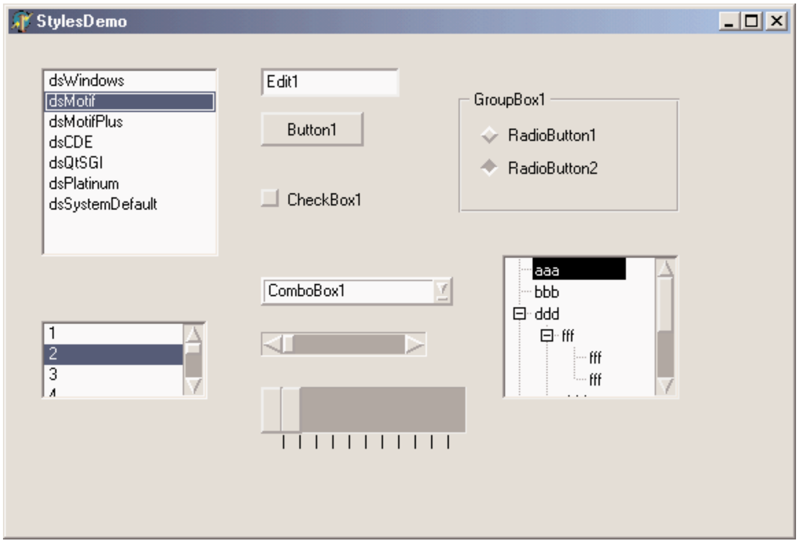
In the StylesDemo program, I've added, among various sample controls, a list box with the names of the default styles, as indicated in the TDefaultStyle enumeration, and this code for its OnDblClick event:

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  Application.Style.DefaultStyle := TDefaultStyle (ListBox1.ItemIndex);
end;
```

The effect is that, by double-clicking the list box, you can change the current application style and immediately see its effect on screen, as demonstrated in Figure 6.15.

FIGURE 6.15:

The StylesDemo program, a Windows application currently with an unusual Motif layout



What's Next?

In this chapter, we have explored the foundations of the libraries available in Delphi for building user interfaces, the native-Windows VCL and the Qt-based CLX. We've discussed the TControl class, its properties, and its most important derived classes.

Then we've started to explore some of the basic components available in Delphi, looking at both libraries. These components correspond to the standard Windows controls and some of the common controls, and they are extremely common in applications. You've also seen how to create main menus and pop-up menus and how to add extra graphics to some of these controls.

The next step, however, is to explore in depth the elements of a complete user interface, discussing other common controls, multipage forms, action lists, and the new Delphi 6 Action Manager, to end up discussing technical details of forms. All of these topics will be covered in the next three chapters.

CHAPTER 7

Advanced VCL Controls

- ListView and TreeView controls
- Multipage forms
- Pages and tabs
- Form-splitting techniques
- Control anchors
- A ToolBar and a StatusBar for the RichEdit control
- Customizing hints

In the preceding chapter, I discussed the core concepts of the TControl class and its derived classes in the VCL and VisualCLX libraries. After that, I provided a sort of rapid tour of the key controls you can use to build a user interface, including editing components, lists, range selectors, and more.

This chapter provides more details on some of these components (such as the ListView and TreeView) and then discusses other controls used to define the overall design of a form, such as the PageControl, TabControl, and Splitter. The chapter also presents examples of splitting forms and resizing controls dynamically. These topics are not particularly complex, but it is worth examining their key concepts briefly.

After these components, I'll introduce toolbars and status bars, including the customization of hints and other slightly more advanced features. This will give us all the foundation material for the following chapter, which covers actions and the new action manager architecture of Delphi 6.

ListView and TreeView Controls

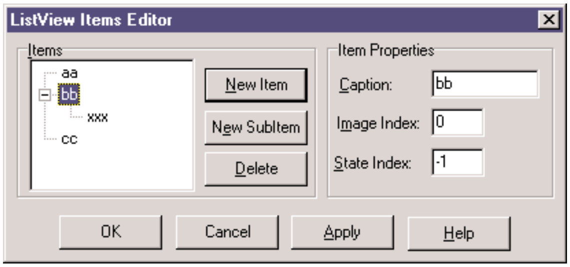
In Chapter 6, I introduced all the various visual controls you can use to display lists of values. The standard list box and combo box components are still very common, but they are often replaced by the more powerful ListView and TreeView controls. Again, these two controls are part of the Win32 common controls, stored in the ComCtl32.DLL library. Similar controls are available in Qt and VisualCLX.

A Graphical Reference List

When you use a ListView component, you can provide bitmaps both indicating the status of the element (for example, the selected item) and describing the contents of the item in a graphical way.

How do we connect the images to a list or tree? We need to refer to the ImageList component we've already used for the images of the menu. A ListView can actually have three image lists: one for the large icons (the LargeImages property), one for the small icons (the SmallImages property), and one used for the state of the items (the StateImages property). In the RefList example on the companion CD, I've set the first two properties using two different ImageList components.

Each of the items of the ListView has an ImageIndex, which refers to its image in the list. For this to work properly, the elements in the two image lists should follow the same order. When you have a fixed image list, you can add items to it using Delphi's ListView Item Editor, which is connected to the Items property. In this editor, you can define items and so-called subitems. The subitems are displayed only in the detailed view (when you set the vsReport value of the `ViewStyle` property) and are connected with the titles set in the `Columns` property.



Caption: bb
Image Index: 0
State Index: -1

In my RefList example (a simple list of references to books, magazines, CD-ROMs, and Web sites), the items are stored to a file, since users of the program can edit the contents of the list, which are automatically saved as the program exits. This way, edits made by the user become persistent. Saving and loading the contents of a ListView is not trivial, since the `TListItems` type doesn't have an automatic mechanism to save the data. As an alternative simple approach, I've copied the data to and from a string list, using a custom format. The string list can then be saved to a file and reloaded with a single command.

The file format is simple, as you can see in the following saving code. For each item of the list, the program saves the caption on one line, the image index on another line (prefixed by the `@` character), and the subitems on the following lines, indented with a tab character:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I, J: Integer;
  List: TStringList;
begin
  // store the items
  List := TStringList.Create;
  try
    for I := 0 to ListView1.Items.Count - 1 do
    begin
      // save the caption
      List.Add (ListView1.Items[I].Caption);
      // save the index
      List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
      // save the subitems (indented)
      for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
        List.Add (#9 + ListView1.Items[I].SubItems [J]);
    end;
    List.SaveToFile (ExtractFilePath (Application.ExeName) + 'Items.txt');
  finally
```

```
    List.Free;
  end;
end;
```

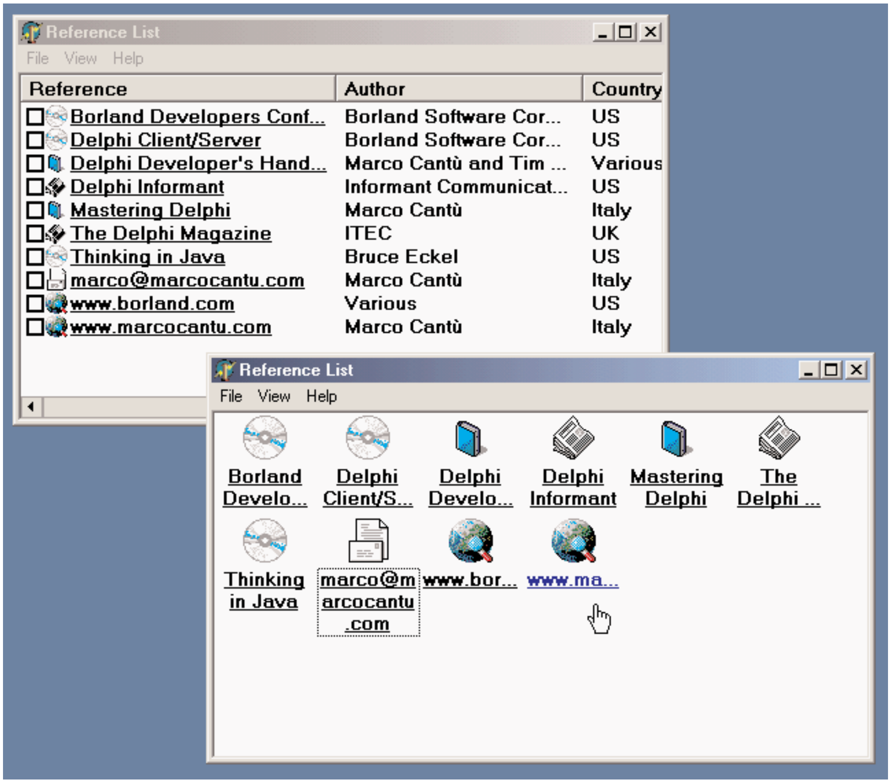
The items are then reloaded in the FormCreate method:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStringList;
  NewItem: TListItem;
  I: Integer;
begin
  // stops warning message
  NewItem := nil;
  // load the items
  ListView1.Items.Clear;
  List := TStringList.Create;
  try
    List.LoadFromFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
    for I := 0 to List.Count - 1 do
      if List [I][1] = #9 then
        NewItem.SubItems.Add (Trim (List [I]))
      else if List [I][1] = '@' then
        NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
      else
      begin
        // a new item
        NewItem := ListView1.Items.Add;
        NewItem.Caption := List [I];
      end;
  finally
    List.Free;
  end;
end;
```

The program has a menu you can use to choose one of the different views supported by the ListView control, and to add check boxes to the items, as in a CheckListBox. You can see some of the various combinations of these styles in Figure 7.1.

FIGURE 7.1:

Different examples of the output of a ListView component of the RefList program, obtained by changing the `ViewStyle` property and adding the check boxes



Reference	Author	Country
□ Borland Developers Conf...	Borland Software Cor...	US
□ Delphi Client/Server	Borland Software Cor...	US
□ Delphi Developer's Hand...	Marco Cantù and Tim ...	Various
□ Delphi Informant	Informant Communicat...	US
□ Mastering Delphi	Marco Cantù	Italy
□ The Delphi Magazine	ITEC	UK
□ Thinking in Java	Bruce Eckel	US
□ marco@marcocantu.com	Marco Cantù	Italy
□ www.borland.com	Various	US
□ www.marcocantu.com	Marco Cantù	Italy

Another important feature, which is common in the detailed or report view of the control, is to let a user sort the items on one of the columns. To accomplish this requires three operations. The first is to set the `SortType` property of the ListView to stBoth or stData. In this way, the ListView will operate the sorting not based on the captions, but by calling the `OnCompare` event for each two items it has to sort. Since we want to do the sorting on each of the columns of the detailed view, we also handle the `OnColumnClick` event (which takes place when the user clicks the column titles in the detailed view, but only if the `ShowColumnHeaders` property is set to True). Each time a column is clicked, the program saves the number of that column in the `nSortCol` private field of the form class:

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  nSortCol := Column.Index;
  ListView1.AlphaSort;
end;
```

Then, in the third step, the sorting code uses either the caption or one of the subitems according to the current sort column:

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem;
  Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
      Item2.SubItems [nSortCol - 1]);
end;
```

The final features I've added to the program relate to mouse operations. When the user left-clicks an item, the RefList program shows a description of the selected item. Right-clicking the selected item sets it in edit mode, and a user can change it (keep in mind that the changes will automatically be saved when the program terminates). Here is the code for both operations, in the OnMouseDown event handler of the ListView control:

```
procedure TForm1.ListView1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
    begin
      // create and show a description
      strDescr := ListView1.Columns [0].Caption + #9 +
        ListView1.Selected.Caption + #13;
      for I := 1 to ListView1.Selected.SubItems.Count do
        strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
          ListView1.Selected.SubItems [I-1] + #13;
      ShowMessage (strDescr);
    end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
end;
```

Although it is not feature-complete, this example shows some of the potential of the ListView control. I've also activated the "hot-tracking" feature, which lets the list view highlight and underline the item under the mouse. The relevant properties of the ListView can be seen in its textual description:

```
object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
    item
      Caption = 'Country'
      Width = 80
    end>
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  FullDrag = True
  HideSelection = False
  HotTrack = True
  HotTrackStyles = [htHandPoint, htUnderlineHot]
  SortType = stBoth
  ViewStyle = vsList
  OnColumnClick = ListView1ColumnClick
  OnCompare = ListView1Compare
  OnMouseDown = ListView1MouseDown
end
```

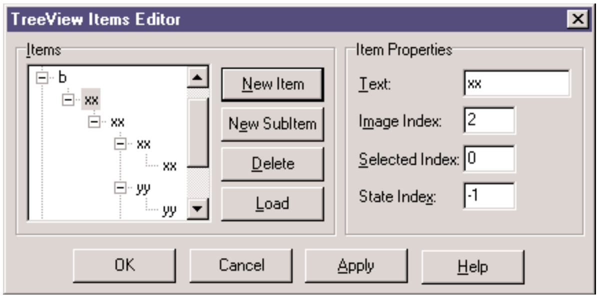
This program is actually quite interesting, and I'll further extend it in Chapter 9, adding a dialog box to it.

To build its CLX version, QRefList, I had to use only one of the image lists, and disable the small images and large images menus, as a ListView is limited to the list and report view styles. Large and small icons are available in a different control, called IconView.

A Tree of Data

Now that we've seen an example based on the ListView, we can examine the TreeView control. The TreeView has a user interface that is flexible and powerful (with support for editing and dragging elements). It is also standard, because it is the user interface of the Windows Explorer. There are properties and various ways to customize the bitmap of each line or of each type of line.

To define the structure of the nodes of the TreeView at design time, you can use the TreeView `Items` property editor. In this case, however, I've decided to load it in the TreeView data at startup, in a way similar to the last example.



Text: xx
Image Index: 2
Selected Index: 0
State Index: -1

The `Items` property of the TreeView component has many member functions you can use to alter the hierarchy of strings. For example, we can build a two-level tree with the following lines:

```
var
  Node: TTreeNode;
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

Using these two methods (`Add` and `AddChild`), we can build a complex structure at run time. But how do we load the information? Again, you can use a StringList at run time, load a text file with the information, and parse it.

However, since the TreeView control has a `LoadFromFile` method, the DragTree and QDragTree examples use the following simpler code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.LoadFromFile (ExtractFilePath (Application.ExeName) +
    'TreeText.txt');
end;
```

The `LoadFromFile` method loads the data in a string list and checks the level of each item by looking at the number of tab characters. (If you are curious, see the `TTreeStrings.GetBufStart` method, which you can find in the ComCtrls unit in the VCL source code included in Delphi.) By the way, the data I've prepared for the TreeView is the organizational chart of a multinational company.

Besides loading the data, the program saves it when it terminates, making the changes persistent. It also has a few menu items to customize the font of the TreeView control and change some other simple settings. The specific feature I've implemented in this example is support for dragging items and entire subtrees. I've set the DragMode property of the component to dmAutomatic and written the event handlers for the OnDragOver and OnDragDrop events.

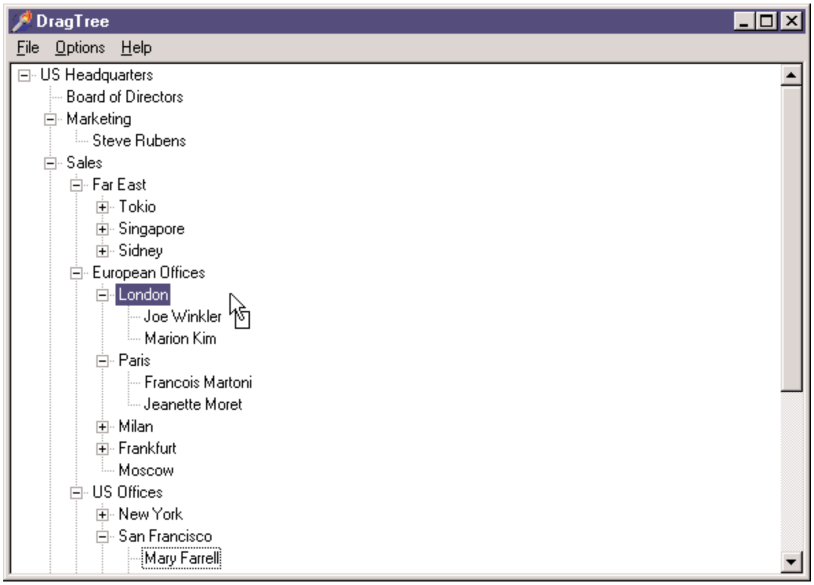
In the first of the two handlers, the program makes sure the user is not trying to drag an item over a child item (which would be moved along with the item, leading to an infinite recursion):

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
  begin
    Accept := True;
    // determines source and target
    SourceNode := TreeView1.Selected;
    // look up the target parent chain
    while (TargetNode.Parent <> nil) and (TargetNode <> SourceNode) do
      TargetNode := TargetNode.Parent;
    // if source is found
    if TargetNode = SourceNode then
      // do not allow dragging over a child
      Accept := False;
  end
  else
    Accept := False;
end;
```

The effect of this code is that (except for the particular case we need to disallow) a user can drag an item of the TreeView over another one, as shown in Figure 7.2. Writing the actual code for moving the items is simple, because the TreeView control provides the support for this operation, through the MoveTo method of the TTreeNode class.

FIGURE 7.2:

The DragTree example during a dragging operation



```
procedure TForm1.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  if TargetNode <> nil then
  begin
    SourceNode := TreeView1.Selected;
    SourceNode.MoveTo (TargetNode, naAddChildFirst);
    TargetNode.Expand (False);
    TreeView1.Selected := TargetNode;
  end;
end;
```

NOTE Among the demos shipping with Delphi is an interesting one showing a custom-draw TreeView control. The example is in the `CustomDraw` subdirectory.

Custom Tree Nodes

Delphi 6 adds a few new features to the TreeView controls, including multiple selection (see the `MultiSelect` and `MultiSelectStyle` properties and the `Selections` array), improved sorting, and several new events.

Another improved area relates to the creation of custom tree node items, which is useful to add extra custom information to each node and possibly create nodes of different classes. To support this technique, there is a new AddNode method for the TTreeItems class and a new specific event, OnCreateNodesClass. In the handler of this event, you return the class of the object to be created, which must inherit from TTreeNode.

As this is a very common technique, I've built an example to discuss it in detail. The CustomNodes example on the CD doesn't focus on a real-world case, but it shows a rather complex situation, in which there are two different custom tree node classes, derived one from the other. The base class adds an ExtraCode property, mapped to virtual methods, and the subclass overrides one of these methods. For the base class the GetExtraCode function simply returns the value, while for the derived class the value is multiplied to the parent node value. Here are the classes and this second method:

```
type
  TMyNode = class (TTreeNode)
  private
    FExtraCode: Integer;
  protected
    procedure SetExtraCode(const Value: Integer); virtual;
    function GetExtraCode: Integer; virtual;
  public
    property ExtraCode: Integer read GetExtraCode write SetExtraCode;
  end;

  TMySubNode = class (TMyNode)
  protected
    function GetExtraCode: Integer; override;
  end;

function TMySubNode.GetExtraCode: Integer;
begin
  Result := fExtraCode * (Parent as TMyNode).ExtraCode;
end;
```

With these custom tree node classes available, the program creates a tree of items, using the first type for the first-level nodes and the second class for the other nodes. As we have only one OnCreateNodeClass event handler, it uses the class reference stored in a private field of the form (CurrentNodeClass of type TTreeNodeClass):

```
procedure TForm1.TreeView1CreateNodeClass(Sender: TCustomTreeView;
  var NodeClass: TTreeNodeClass);
begin
  NodeClass := CurrentNodeClass;
end;
```

The program sets this class reference before creating nodes of each type—for example, with code like

```
var
  MyNode: TMyNode;
begin
  CurrentNodeClass := TMyNode;
  MyNode := TreeView1.Items.AddChild (nil, 'item' + IntToStr (nValue))
    as TMyNode;
  MyNode.ExtraCode := nValue;
```

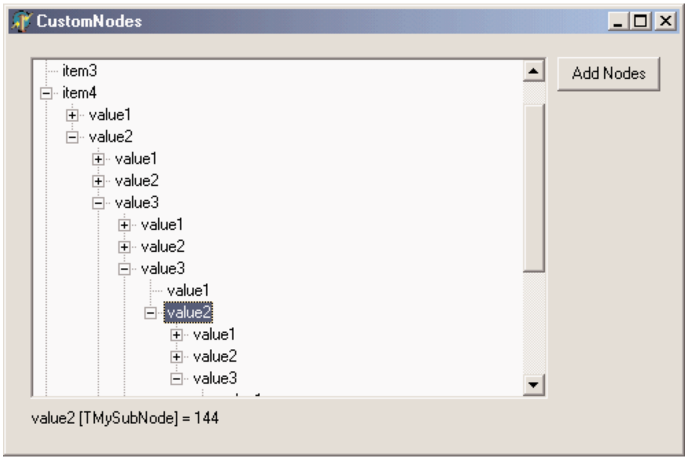
Once the entire tree has been created, as the user selects an item, you can cast its type to TMyNode and access to the extra properties (but also methods and data):

```
procedure TForm1.TreeView1Click(Sender: TObject);
var
  MyNode: TMyNode;
begin
  MyNode := TreeView1.Selected as TMyNode;
  Label1.Caption := MyNode.Text + ' [' + MyNode.ClassName + '] = ' +
    IntToStr (MyNode.ExtraCode);
end;
```

This is the code used by the CustomNodes example to display the description of the selected node into a label, as you can see in Figure 7.3. Note that when you select an item down into the tree, its value is multiplied for that of each of the parent nodes. Though there are certainly easier ways to obtain this effect, having a tree view with item objects created from different classes of a hierarchy provides an object-oriented structure upon which you can base some very complex code.

FIGURE 7.3:

The CustomNodes example has a tree view with node objects based on different custom classes, thanks to the new OnCreateNodesClass event.



value2 [TMySubNode] = 144

Multiple-Page Forms

When you have a lot of information and controls to display in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages.

There are two controls you can use to build a multiple-page application in Delphi:

- You can use the PageControl component, which has tabs on one of the sides and multiple pages (similar to panels) covering the rest of its surface. As there is one page per tab, you can simply place components on each page to obtain the proper effect both at design time and at run time.
- You can use the TabControl, which has only the tab portion but offers no pages to host the information. In this case, you'll want to use one or more components to mimic the *page change* operation.

A third related class, the `TabSheet`, represents a single page of the PageControl. This is not a stand-alone component and is not available on the Component Palette. You create a `TabSheet` at design time by using the local menu of the PageControl or at run time by using methods of the same control.

NOTE Delphi still includes the Notebook, TabSet, and TabbedNotebook components introduced in early versions. Use these components only if you need to create a 16-bit version of an application. For any other purpose, the PageControl and TabControl components, which encapsulate Win32 common controls, provide a more modern user interface. Actually, in 32-bit versions of Delphi, the TabbedNotebook component was reimplemented using the Win32 PageControl internally, to reduce the code size and update the look.

PageControls and TabSheets

As usual, instead of duplicating the Help system's list of properties and methods of the PageControl component, I've built an example that stretches its capabilities and allows you to change its behavior at run time. The example, called Pages, has a PageControl with three pages. The structure of the PageControl and of the other key components is listed here:

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
```

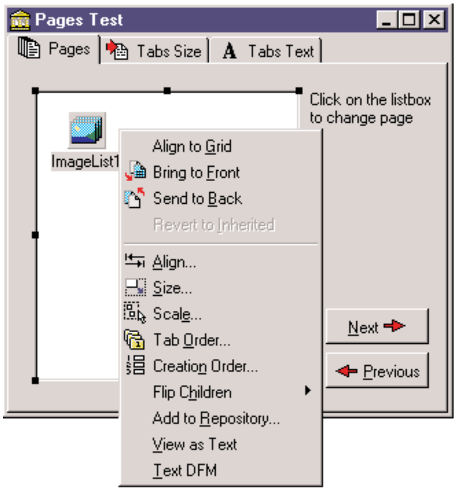
```
      HotTrack = True
      Images = ImageList1
      MultiLine = True
      object TabSheet1: TTabSheet
        Caption = 'Pages'
        object Label3: TLabel
        object ListBox1: TListBox
      end
      object TabSheet2: TTabSheet
        Caption = 'Tabs Size'
        ImageIndex = 1
        object Label1: TLabel
        // other controls
      end
      object TabSheet3: TTabSheet
        Caption = 'Tabs Text'
        ImageIndex = 2
        object Memo1: TMemo
          Anchors = [akLeft, akTop, akRight, akBottom]
          OnChange = Memo1Change
        end
        object BitBtnChange: TBitBtn
          Anchors = [akTop, akRight]
          Caption = '&Change'
        end
      end
    end
    object BitBtnPrevious: TBitBtn
      Anchors = [akRight, akBottom]
      Caption = '&Previous'
      OnClick = BitBtnPreviousClick
    end
    object BitBtnNext: TBitBtn
      Anchors = [akRight, akBottom]
      Caption = '&Next'
      OnClick = BitBtnNextClick
    end
    object ImageList1: TImageList
      Bitmap = {...}
    end
  end
```

Notice that the tabs are connected to the bitmaps provided by an ImageList control and that some controls use the `Anchors` property to remain at a fixed distance from the right or bottom borders of the form. Even if the form doesn't support resizing (this would have been far too complex to set up with so many controls), the positions can change when the tabs are displayed on multiple lines (simply increase the length of the captions) or on the left side of the form.

Each TabSheet object has its own `Caption`, which is displayed as the sheet's tab. At design time, you can use the local menu to create new pages and to move between pages. You can see the local menu of the PageControl component in Figure 7.4, together with the first page. This page holds a list box and a small caption, and it shares two buttons with the other pages.

FIGURE 7.4:

The first sheet of the PageControl of the Pages example, with its local menu



If you place a component on a page, it is available only in that page. How can you have the same component (in this case, two bitmap buttons) in each of the pages, without duplicating it? Simply place the component on the form, outside of the PageControl (or before aligning it to the client area) and then move it in front of the pages, calling the Bring To Front command of the form's local menu. The two buttons I've placed in each page can be used to move back and forth between the pages and are an alternative to using the tabs. Here is the code associated with one of them:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.SelectNextPage (True);
end;
```

The other button calls the same procedure, passing False as its parameter to select the previous page. Notice that there is no need to check whether we are on the first or last page, because the `SelectNextPage` method considers the last page to be the one before the first and will move you directly between those two pages.

Now we can focus on the first page again. It has a list box, which at run time will hold the names of the tabs. If a user clicks an item of this list box, the current page changes. This is the third method available to change pages (after the tabs and the Next and Previous buttons). The list box is filled in the `FormCreate` method, which is associated with the `OnCreate` event of the form and copies the caption of each page (the `Page` property stores a list of TabSheet objects):

```
for I := 0 to PageControl1.PageCount - 1 do
  ListBox1.Items.Add (PageControl1.Pages.Caption);
```

When you click a list item, you can select the corresponding page:

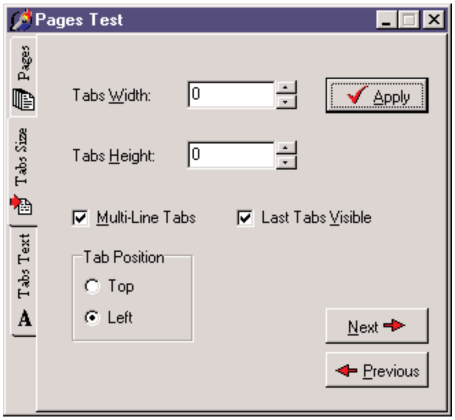
```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage := PageControl1.Pages [ListBox1.ItemIndex];
end;
```

The second page hosts two edit boxes (connected with two UpDown components), two check boxes, and two radio buttons, as you can see in Figure 7.5. The user can input a number (or choose it by clicking the up or down buttons with the mouse or pressing the Up or Down arrow key while the corresponding edit box has the focus), check the boxes and the radio buttons, and then click the Apply button to make the changes:

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
  // set tab width, height, and lines
  PageControl1.TabWidth := StrToInt (EditWidth.Text);
  PageControl1.TabHeight := StrToInt (EditHeight.Text);
  PageControl1.MultiLine := CheckBoxMultiLine.Checked;
  // show or hide the last tab
  TabSheet3.TabVisible := CheckBoxVisible.Checked;
  // set the tab position
  if RadioButton1.Checked then
    PageControl1.TabPosition := tpTop
  else
    PageControl1.TabPosition := tpLeft;
end;
```

FIGURE 7.5:

The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control.


Pages Test

Tabs Width: 0

Tabs Height: 0

☑ Multi-Line Tabs ☑ Last Tabs Visible

Tab Position
○ Top
● Left

With this code, we can change the width and height of each tab (remember that 0 means the size is computed automatically from the space taken by each string). We can choose to have either multiple lines of tabs or two small arrows to scroll the tab area, and move them to the left side. The control also allows tabs to be placed on the bottom or on the right, but our program doesn't allow that, because it would make the placement of the other controls quite complex.

You can also hide the last tab on the PageControl, which corresponds to the TabSheet3 component. If you hide one of the tabs by setting its TabVisible property to False, you cannot reach that tab by clicking on the Next and Previous buttons, which are based on the SelectNextPage method. Instead, you should use the FindNextPage function, as shown below in this new version of the Next button's OnClick event handler:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.ActivePage := PageControl1.FindNextPage (
    PageControl1.ActivePage, True, False);
end;
```

The last page has a memo component, again with the names of the pages (added in the FormCreate method). You can edit the names of the pages and click the Change button to change the text of the tabs, but only if the number of strings matches the number of tabs:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
  if Memo1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please', mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption := Memo1.Lines [I];
  BitBtnChange.Enabled := False;
end;
```

Finally the last button, Add Page, allows you to add a new tab sheet to the page control, although the program doesn't add any components to it. The (empty) tab sheet object is created using the page control as its owner, but it won't work unless you also set the PageControl property. Before doing this, however, you should make the new tab sheet visible. Here is the code:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);
var
  strCaption: string;
  NewTabSheet: TTabSheet;
begin
  strCaption := 'New Tab';
  if InputQuery ('New Tab', 'Tab Caption', strCaption) then
  begin
    // add a new empty page to the control
    NewTabSheet := TTabSheet.Create (PageControl1);
    NewTabSheet.Visible := True;
    NewTabSheet.Caption := strCaption;
    NewTabSheet.PageControl := PageControl1;
    PageControl1.ActivePage := NewTabSheet;
    // add it to both lists
    Memo1.Lines.Add (strCaption);
    ListBox1.Items.Add (strCaption);
  end;
end;
```

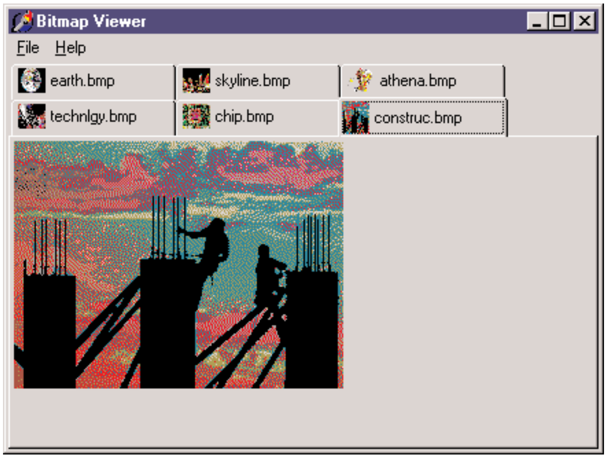
TIP Whenever you write a form based on a PageControl, remember that the first page displayed at run time is the page you were in before the code was compiled. This means that if you are working on the third page and then compile and run the program, it will start with that page. A common way to solve this problem is to add a line of code in the `FormCreate` method to set the PageControl or notebook to the first page. This way, the current page at design time doesn't determine the initial page at run time.

An Image Viewer with Owner-Draw Tabs

The use of the TabControl and of a dynamic approach, as described in the last example, can also be applied in more general (and simpler) cases. Every time you need multiple pages that all have the same type of content, instead of replicating the controls in each page, you can use a TabControl and change its contents when a new tab is selected.

This is what I'll do in the multiple-page bitmap viewer example, called BmpViewer. The image that appears in the TabControl of this form, aligned to the whole client area, depends on the selection in the tab above it (as you can see in Figure 7.6).

FIGURE 7.6: The interface of the bitmap viewer in the BmpViewer example. Notice the owner-draw tabs.



At the beginning, the TabControl is empty. After selecting File ➢ Open, the user can choose various files in the File Open dialog box, and the array of strings with the names of the files (the `Files` property of the `OpenDialog1` component) is added to the tabs (the `Tabs` property of `TabControl1`):

```
procedure TFormBmpViewer.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    TabControl1.Tabs.AddStrings (OpenDialog1.Files);
    TabControl1.TabIndex := 0;
    TabControl1Change (TabControl1);
  end;
end;
```

After we display the new tabs, we have to update the image so that it matches the first tab. To accomplish this, the program calls the method connected with the `OnChange` event of the TabControl, which loads the file corresponding to the current tab in the image component:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (TabControl1.Tabs [TabControl1.TabIndex]);
end;
```

This example works, unless you select a file that doesn't contain a bitmap. The program will warn the user with a standard exception, ignore the file, and continue its execution.

The program also allows pasting the bitmap on the clipboard (without actually copying it, though) and copying the current bitmap to it. Clipboard support is available in Delphi via the global `Clipboard` object defined in the ClipBrd unit. For copying or pasting bitmaps, you can use the `Assign` method of the `TClipboard` and `TBitmap` classes. When you select the Edit ➢ Paste command of the example, a new tab named Clipboard is added to the tab set (unless it is already present). Then the number of the new tab is used to change the active tab:

```
procedure TFormBmpViewer.Paste1Click(Sender: TObject);
var
  TabNum: Integer;
begin
  // try to locate the page
  TabNum := TabControl1.Tabs.IndexOf ('Clipboard');
  if TabNum < 0 then
    // create a new page for the Clipboard
    TabNum := TabControl1.Tabs.Add ('Clipboard');
  // go to the Clipboard page and force repaint
  TabControl1.TabIndex := TabNum;
  TabControl1Change (Self);
end;
```

The Edit ➢ Copy operation, instead, is as simple as copying the bitmap currently in the image control:

```
Clipboard.Assign (Image1.Picture.Graphic);
```

To account for the possible presence of the Clipboard tab, the code of the TabControl1Change method becomes:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
var
  TabText: string;
begin
  Image1.Visible := True;
  TabText := TabControl1.Tabs [TabControl1.TabIndex];
  if TabText <> 'Clipboard' then
    // load the file indicated in the tab
    Image1.Picture.LoadFromFile (TabText)
  else
    {if the tab is 'Clipboard' and a bitmap
    is available in the clipboard}
    if Clipboard.HasFormat (cf_Bitmap) then
      Image1.Picture.Assign (Clipboard)
    else
    begin
      // else remove the clipboard tab
      TabControl1.Tabs.Delete (TabControl1.TabIndex);
      if TabControl1.Tabs.Count = 0 then
        Image1.Visible := False;
    end;
```

This program pastes the bitmap from the Clipboard each time you change the tab. The program stores only one image at a time, and it has no way to store the Clipboard bitmap. However, if the Clipboard content changes and the bitmap format is no longer available, the Clipboard tab is automatically deleted (as you can see in the listing above). If no more tabs are left, the Image component is hidden.

An image can also be removed using either of two menu commands: Cut or Delete. Cut removes the tab after making a copy of the bitmap to the Clipboard. In practice, the Cut1Click method does nothing besides calling the Copy1Click and Delete1Click methods. The Copy1Click method is responsible for copying the current image to the Clipboard, Delete1Click simply removes the current tab. Here is their code:

```
procedure TFormBmpViewer.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign (Image1.Picture.Graphic);
end;
```

```
procedure TFormBmpViewer.Delete1Click(Sender: TObject);
begin
  with TabControl1 do
  begin
    if TabIndex >= 0 then
      Tabs.Delete (TabIndex);
    if Tabs.Count = 0 then
      Image1.Visible := False;
  end;
end;
```

One of the special features of the example is that the TabControl has the OwnerDraw property set to True. This means that the control won't paint the tabs (which will be empty at design time) but will have the application do this, by calling the OnDrawTab event. In its code, the program displays the text vertically centered, using the DrawText API function. The text displayed is not the entire file path but only the filename. Then, if the text is not *None*, the program reads the bitmap the tab refers to and paints a small version of it in the tab itself. To accomplish this, the program uses the TabBmp object, which is of type TBitmap and is created and destroyed along with the form. The program also uses the BmpSide constant to position the bitmap and the text properly:

```
procedure TFormBmpViewer.TabControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
 TabText: string;
 OutRect: TRect;
begin
  TabText := TabControl1.Tabs [TabIndex];
  OutRect := Rect;
  InflateRect (OutRect, -3, -3);
  OutRect.Left := OutRect.Left + BmpSide + 3;
  DrawText (Control.Canvas.Handle, PChar (ExtractFileName (TabText)),
    Length (ExtractFileName (TabText)), OutRect,
    dt_Left or dt_SingleLine or dt_VCenter);
  if TabText = 'Clipboard' then
    if Clipboard.HasFormat (cf_Bitmap) then
      TabBmp.Assign (Clipboard)
    else
      TabBmp.FreeImage
  else
    TabBmp.LoadFromFile (TabText);
  OutRect.Left := OutRect.Left - BmpSide - 3;
  OutRect.Right := OutRect.Left + BmpSide;
  Control.Canvas.StretchDraw (OutRect, TabBmp);
end;
```

The program has also support for printing the current bitmap, after showing a page preview form in which the user can select the proper scaling. This extra portion of the program I built for earlier editions of the book is not discussed in detail, but I've left the code in the program so that you can have a look at its code anyway.

The User Interface of a Wizard

Just as you can use a TabControl without pages, you can also take the opposite approach and use a PageControl without tabs. What I want to focus on now is the development of the user interface of a wizard. In a wizard, you are directing the user through a sequence of steps, one screen at a time, and at each step you typically want to offer the choice of proceeding to the next step or going back to correct input entered in a previous step. So instead of tabs that can be selected in any order, wizards typically offer Next and Back buttons to navigate. This won't be a complex example; its purpose is just to give you a few guidelines. The example is called WizardUI.

The starting point is to create a series of pages in a PageControl and set the `TabVisible` property of each TabSheet to False (while keeping the `Visible` property set to True). Unlike past versions, since Delphi 5 you can also hide the tabs at design time. In this case, you'll need to use the shortcut menu of the page control or the combo box of the Object Inspector to move to another page, instead of the tabs. But why don't you want to see the tabs at design time? You can place controls on the pages and then place extra controls in front of the pages (as I've done in the example), without seeing their relative positions change at run time. You might also want to remove the useless captions of the tabs, which take up space in memory and in the resources of the application.

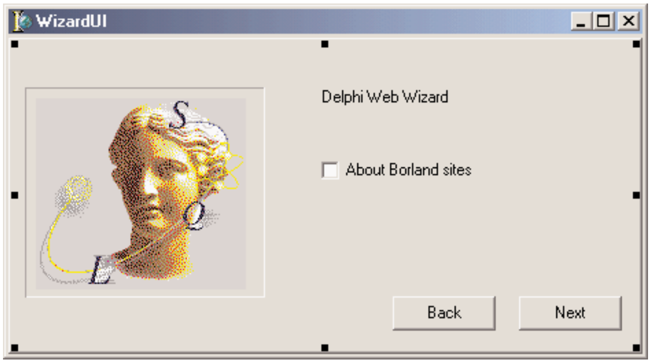
In the first page, I've placed on one side an image and a bevel control and on the other side some text, a check box, and two buttons. Actually, the Next button is inside the page, while the Back button is over it (and is shared by all the pages). You can see this first page at design time in Figure 7.7. The following pages look similar, with a label, check boxes, and buttons on the right side and nothing on the left.

When you click the Next button on the first page, the program looks at the status of the check box and decides which page is the following one. I could have written the code like this:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  BtnBack.Enabled := True;
  if CheckInprise.Checked then
    PageControl1.ActivePage := TabSheet2
  else
    PageControl1.ActivePage := TabSheet3;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

FIGURE 7.7:

The first page of the WizardUI example at design time



After enabling the common Back button, the program changes the active page and finally moves the graphical portion to the new page. Because this code has to be repeated for each button, I've placed it in a method after adding a couple of extra features. This is the actual code:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  if CheckInprise.Checked then
    MoveTo (TabSheet2)
  else
    MoveTo (TabSheet3);
end;

procedure TForm1.MoveTo(TabSheet: TTabSheet);
begin
  // add the last page to the list
  BackPages.Add (PageControl1.ActivePage);
  BtnBack.Enabled := True;
  // change page
  PageControl1.ActivePage := TabSheet;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

Besides the code I've already explained, the MoveTo method adds the last page (the one before the page change) to a list of visited pages, which behaves like a stack. In fact, the BackPages object of the TList class is created as the program starts and the last page is always added to the end. When the user clicks the Back button, which is not dependent on the page, the program extracts the last page from the list, deletes its entry, and moves to that page:

```
procedure TForm1.btnBackClick(Sender: TObject);
var
```

```
    LastPage: TTabSheet;
  begin
    // get the last page and jump to it
    LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
    PageControl1.ActivePage := LastPage;
    // delete the last page from the list
    BackPages.Delete (BackPages.Count - 1);
    // eventually disable the back button
    BtnBack.Enabled := not (BackPages.Count = 0);
    // move image and bevel
    Bevel1.Parent := PageControl1.ActivePage;
    Image1.Parent := PageControl1.ActivePage;
  end;
```

With this code, the user can move back several pages until the list is empty, at which point we disable the Back button. The complication we need to deal with is that while moving from a particular page, we know which pages are its "next" and "previous," but we don't know which page we came from, because there can be multiple paths to reach a page. Only by keeping track of the movements with a list can we reliably go back.

The rest of the code of the program, which simply shows some Web site addresses, is very simple. The good news is that you can reuse the navigational structure of this example in your own programs and modify only the graphical portion and the content of the pages. Actually, as most of the labels of the programs show HTTP addresses, a user can click those labels to open the default browser showing that page. This is accomplished by extracting the HTTP address from the label and calling the ShellExecute function.

```
  procedure TForm1.LabelLinkClick(Sender: TObject);
  var
    Caption, StrUrl: string;
  begin
    Caption := (Sender as TLabel).Caption;
    StrUrl := Copy (Caption, Pos ('http://', Caption), 1000);
    ShellExecute (Handle, 'open', PChar (StrUrl), '', '', sw_Show);
  end;
```

The method above is hooked to the OnClick event of many labels of the form, which have been turned into *links* by setting its Cursor to a hand. This is one of the labels:

```
  object Label2: TLabel
    Cursor = crHandPoint
    Caption = 'Main site: http://www.borland.com'
    OnClick = LabelLinkClick
  end
```

Form-Splitting Techniques

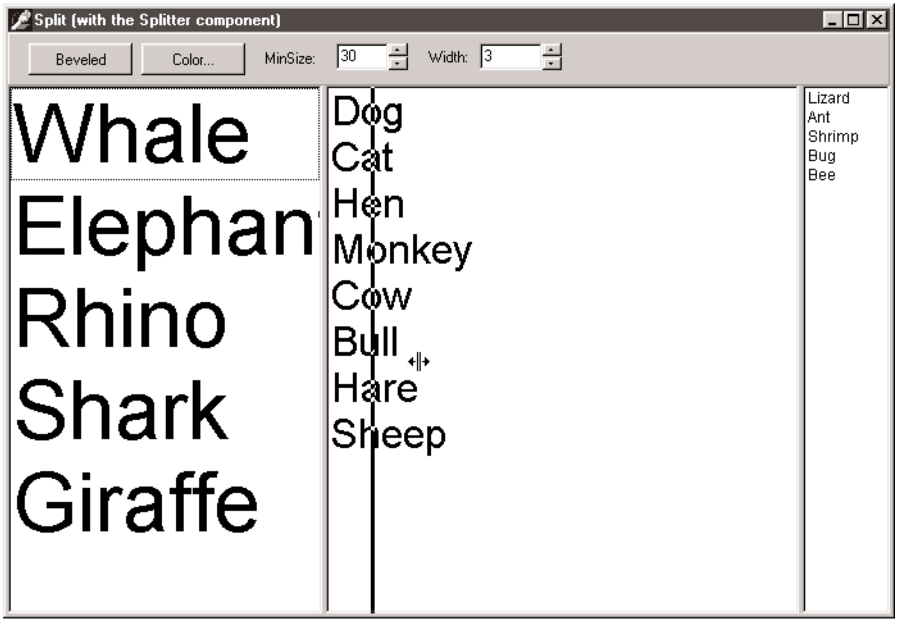
There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the `Constraints` property of the controls it relates to. As we'll see in the Split1 example, this allows us to define maximum and minimum positions of the splitter and of the form.

To build this example, simply place a ListBox component in a form; then add a Splitter component, a second ListBox, another Splitter, and finally a third ListBox component. The form also has a simple toolbar based on a panel.

By simply placing these two splitter components, you give your form the complete functionality of moving and sizing the controls it hosts at run time. The `Width`, `Beveled`, and `Color` properties of the splitter components determine their appearance, and in the Split1 example you can use the toolbar controls to change them. Another relevant property is `MinSize`, which determines the minimum size of the components of the form. During the splitting operation (see Figure 7.8), a line marks the final position of the splitter, but you cannot drag this line beyond a certain limit. The behavior of the Split1 program is not to let controls become too small. An alternative technique is to set the new `AutoSnap` property of the splitter to True. This property will make the splitter hide the control when its size goes below the `MinSize` limit.

FIGURE 7.8:

The splitter component of the Split1 example determines the minimum size for each control on the form, even those not adjacent to the splitter itself.



I suggest you try using the Split1 program, so that you'll fully understand how the splitter affects its adjacent controls and the other controls of the form. Even if we set the MinSize property, a user of this program can reduce the size of its entire form to a minimum, hiding some of the list boxes. If you test the Split2 version of the example, instead, you'll get better behavior. In Split2, I've set some Constraints for the ListBox controls—for example,

```
object ListBox1: TListBox
  Constraints.MaxHeight = 400
  Constraints.MinHeight = 200
  Constraints.MinWidth = 150
```

The size constraints are applied only as you actually resize the controls, so to make this program work in a satisfactory way, you have to set the ResizeStyle property of the two splitters to rsUpdate. This value indicates that the position of the controls is updated for every movement of the splitter, not only at the end of the operation. If you select the rsLine or the new rsPattern values, instead, the splitter simply draws a line in the required position, checking the MinSize property but not the constraints of the controls.

TIP

When you set the Splitter component's AutoSnap property to True, the splitter will completely hide the neighboring control when the size of that control is below the minimum set for it in the Splitter component.

Horizontal Splitting

The Splitter component can also be used for horizontal splitting, instead of the default vertical splitting. However, this approach is a little more complicated. Basically you can place a component on a form, align it to the top, and then place the splitter on the form. By default, it will be left aligned. Choose the alTop value for the Align property, and then resize the component manually, by changing the Height property in the Object Inspector (or by resizing the component).

You can see a form with a horizontal splitter in the SplitH example. This program has two memo components you can open a file into, and it has a splitter dividing them, defined as:

```
object Splitter1: TSplitter
  Cursor = crVSplit
  Align = alTop
  OnMoved = Splitter1Moved
end
```

When you double-click a memo, the program loads a text file into it (notice the structure of the with statement):

```
procedure TForm1.MemoDblClick(Sender: TObject);
begin
```

```
    with Sender as TMemo, OpenDialog1 do
      if Execute then
        Lines.LoadFromFile (FileName);
  end;
```

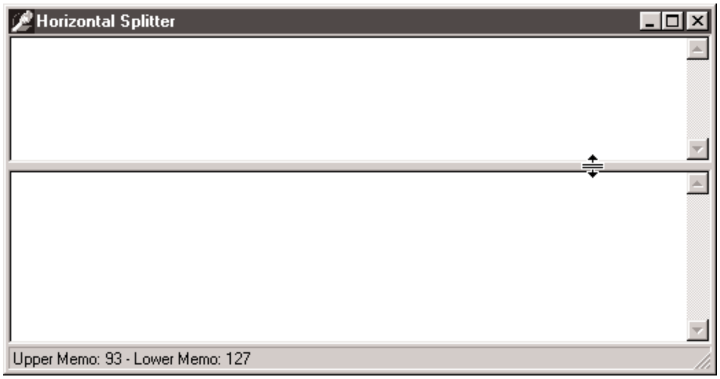
The program features a status bar, which keeps track of the current height of the two memo components. It handles the `OnMoved` event of the splitter (the only event of this component) to update the text of the status bar. The same code is executed whenever the form is resized:

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format ('Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

You can see the effect of this code by looking at Figure 7.9, or by running the SplitH example.

FIGURE 7.9:

The status bar of the SplitH example indicates the position of the horizontal splitter component.



Upper Memo: 93 - Lower Memo: 127

Splitting with a Header

An alternative to using splitters is to use the standard HeaderControl component. If you place this control on a form, it will be automatically aligned with the top of the form. Then you can add the three list boxes to the rest of the client area of the form. The first list box can be aligned on the left, but this time you cannot align the second and third list box as well. The problem is that the sections of the header can be dragged outside the visible surface of the form. If the list boxes use automatic alignment, they cannot move outside the visible surface of the form, as the program requires.

The solution is to define the sections of the header, using the specific editor of the `Sections` property. This property editor allows you to access the various subobjects of the collection, changing various settings. You can set the caption and alignment of the text; the current, minimum, and maximum size of the header; and so on. Setting the limit values is a powerful tool, and it replaces the MinSize property of the splitter or the constraints of the list boxes we've used in past examples. You can see the output of this program, named HdrSplit, in Figure 7.10.

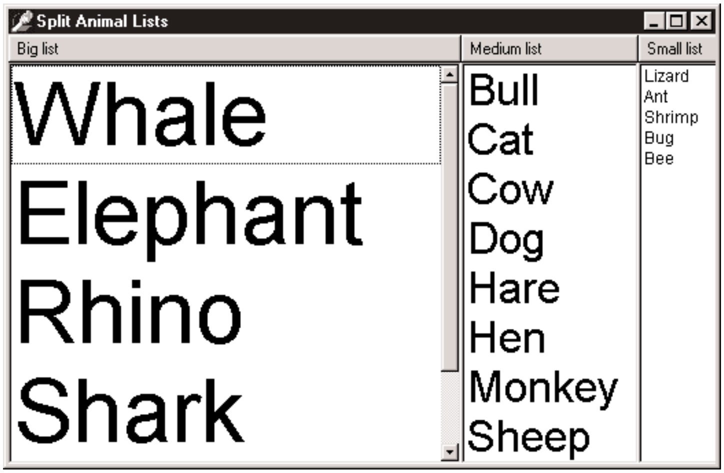


FIGURE 7.10: The output of the HdrSplit example

We need to handle two events: OnSectionResize and OnSectionClick. The first handler simply resizes the list box connected with the modified section (determined by associating numbers with the ImageIndex property of each section and using it to determine the name of the list box control):

```
procedure TForm1.HeaderControl1SectionResize(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (Section.ImageIndex))
    as TListBox;
  List.Width := Section.Width;
end;
```

Along with this event, we need to handle the resizing of the form, using it to synchronize the list boxes with the sections, which are all resized by default:

```
procedure TForm1.FormResize(Sender: TObject);
var
  I: Integer;
  List: TListBox;
begin
  for I := 0 to 2 do
  begin
    List := FindComponent ('ListBox' + IntToStr (
      HeaderControl1.Sections[I].ImageIndex)) as TListBox;
    List.Left := HeaderControl1.Sections[I].Left;
    List.Width := HeaderControl1.Sections[I].Width;
  end;
end;
```

After setting the height of the list boxes, this method simply calls the previous one, passing parameters that we won't use in this example. The second method of the HeaderControl, called in response to a click on one of the sections, is used to sort the contents of the corresponding list box:

```
procedure TForm1.HeaderControl1SectionClick(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (Section.ImageIndex))
    as TListBox;
  List.Sorted := not List.Sorted;
end;
```

Of course, this code doesn't provide the common behavior of sorting the elements when you click the header and then sorting them in the reverse order if you click again. To implement this, you should write your own sorting algorithm.

Finally, the HdrSplit example uses a new feature for the header control. It sets the `DragReorder` property to enable dragging operations to reorder the header sections. When this operation is performed, the control fires the `OnSectionDrag` event, where you can exchange the positions of the list boxes. This event fires before the sections are actually moved, so I have to use the coordinates of the other section:

```
procedure TForm1.HeaderControl1SectionDrag(Sender: TObject; FromSection,
  ToSection: THeaderSection; var AllowDrag: Boolean);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (FromSection.ImageIndex))
    as TListBox;
  List.Left := ToSection.Left;
  List.Width := ToSection.Width;

  List := FindComponent ('ListBox' + IntToStr (ToSection.ImageIndex))
    as TListBox;
  List.Left := FromSection.Left;
  List.Width :=fromSection.Width
end;
```

Control Anchors

In this chapter, I've described how you can use alignment and splitters to create nice, flexible user interfaces, that adapt to the current size of the form, giving users maximum freedom. Delphi also supports right and bottom anchors. Before this feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom, unless it was aligned to the bottom or right sides. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to have a button anchored to the bottom-right corner of the form, place the button in the required position and set its `Anchors` property to `[akRight, akBottom]`. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

On the other hand, if you place a large component such as a Memo or a ListBox in the middle of a form, you can set its `Anchors` property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

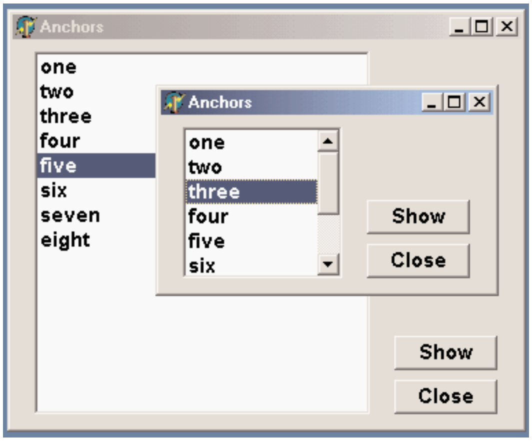
TIP Anchors, like constraints, work both at design time and at run time, so you should set them up as early as possible, to benefit from this feature while you're designing the form as well as at run time.

As an example of both approaches, you can try out the Anchors application, which has two buttons on the bottom-right corner and a list box in the middle. As shown in Figure 7.11, the controls automatically move and stretch as the form size changes. To make this form work properly, you must also set its `Constraints` property; otherwise, as the form becomes too small the controls can overlap or disappear.

TIP If you remove all of the anchors, or two opposite ones (for example, left and right), the resize operations will cause the control to float. The control keeps its current size, and the system adds or removes the same number of pixels on each side of it. This can be defined as a centered anchor, because if the component is initially in the middle of the form it will keep that position. In any case, if you want a centered control, you should generally use both opposite anchors, so that if the user makes the form larger, the control size will grow as well. In the case just presented, in fact, making the form larger leaves a small control in its center.

FIGURE 7.11:

The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only a proper use of the `Anchors` property.



The ToolBar Control

In early versions of Delphi, toolbars had to be created using panels and speed buttons. Starting with version 3, Delphi introduced a specific ToolBar component, which encapsulates the corresponding Win32 common control or the corresponding Qt widget in VisualCLX. This component provides a toolbar, with its own buttons, and it has many advanced capabilities. To use this component, you place it on a form and then use the component editor (the shortcut menu activated by a right mouse button click) to create a few buttons and separators.

Building a Toolbar with a Panel

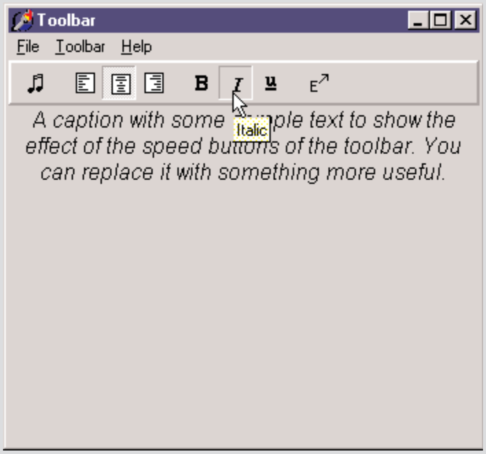
Before the toolbar control was available in Delphi, the standard approach for building a toolbar was to use a panel aligned to the top of the form and place SpeedButton components inside it. A *speed button* is a lightweight graphical control (consuming no Windows resources); it cannot receive the input focus, it has no tab order, and it is faster to create and paint than a bitmap button.

Speed buttons can behave like push buttons, check boxes, or radio buttons, and they can have different bitmaps depending on their status. To make a group of speed buttons work like radio buttons, just place some speed buttons on the panel, select all of them, and give the same value to each one's `GroupIndex` property. All the buttons having the same `GroupIndex` become mutually exclusive selections. One of these buttons should always be selected, so remember to set the `Down` property to True for one of them at design time or as soon as the program starts.

Continued on next page

By setting the `AllowAllUp` property, you can create a group of mutually exclusive buttons, each of which can be *up*—that is, a group from which the user can select one option or leave them all unselected. As a special case, you can make a speed button work as a check box, simply by defining a group (the `GroupIndex` property) that has only one button and that allows it to be deselected (the `AllowAllUp` property).

Finally, you can set the `Flat` property of all the SpeedButton components to True, obtaining a more modern user interface. If you are interested in this approach, you can look at the PanelBar example, illustrated here:



A caption with some sample text to show the effect of the speed buttons of the toolbar. You can replace it with something more useful.

The use of SpeedButton controls is becoming less common. Besides the fact that the ToolBar control is very handy and definitely more standard, speed buttons have two big problems. First, each of them requires a specific bitmap and cannot use one from an image list (unless you write some complex code). Second, speed buttons don't work very well with actions, because some properties, such as the Down state, do not map directly.

The toolbar is populated with objects of the `TToolButton` class. These objects have a fundamental property, `Style`, which determines their behavior:

- The tbsButton style indicates a standard push button.
- The tbsCheck style indicates a button with the behavior of a check box, or that of a radio button if the button is `Grouped` with the others in its block (determined by the presence of separators).

- The tbsDropDown style indicates a drop-down button, a sort of combo box. The drop-down portion can be easily implemented in Delphi by connecting a PopupMenu control to the `DropdownMenu` property of the control.
- The tbsSeparator and tbsDivider styles indicate separators with no or different vertical lines (depending on the `Flat` property of the toolbar).

To create a graphic toolbar, you can add an ImageList component to the form, load some bitmaps into it, and then connect the ImageList with the `Images` property of the toolbar. By default, the images will be assigned to the buttons in the order they appear, but you can change this quite easily by setting the `ImageIndex` property of each toolbar button. You can prepare further ImageLists for special conditions of the buttons and assign them to the `DisabledImages` and `HotImages` properties of the toolbar. The first group is used for the disabled buttons; the second for the button currently under the mouse.

NOTE In a nontrivial application, you would generally create toolbars using an ActionList or the new Action Manager architecture, both discussed in the next chapter. In this case, you'll attach very little behavior to the toolbar buttons, as their properties and events will be managed by the action components.

The RichBar Example

As an example of the use of a toolbar, I've built the RichBar application, which has a RichEdit component you can operate by using the toolbar. The program has buttons for loading and saving files, for copy and paste operations, and to change some of the attributes of the current font.

I don't want to cover the details of the features of the RichEdit control, which are many, nor discuss the details of this application, which has quite a lot of code. All I want to do is to focus on features specific to the ToolBar used by the example and visible in Figure 7.12. This toolbar has buttons, separators, and even a drop-down menu and two combo boxes discussed in the next section.

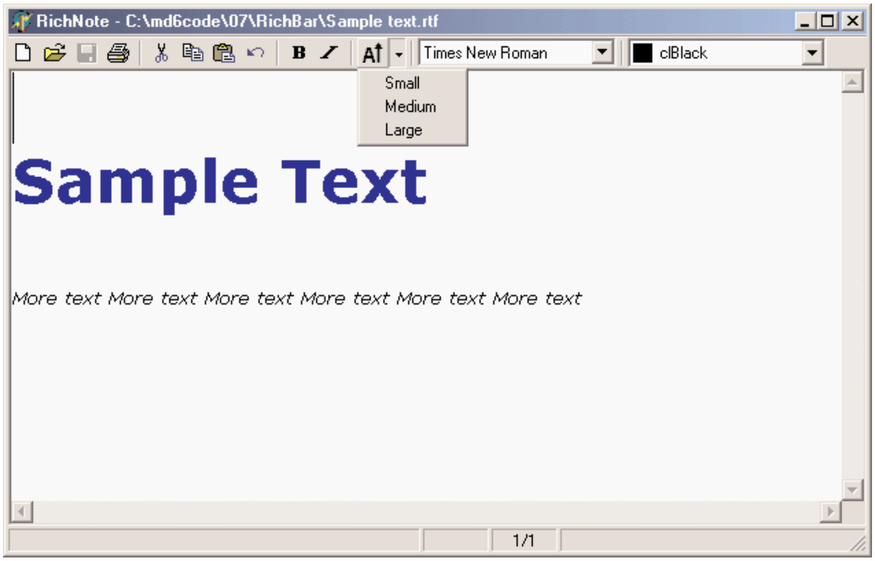
The various buttons implement features, one of them being a complete scheme for opening and saving the text files, including the ability to ask the user to save any modified file before opening a new one, to avoid losing any changes. The file-handling portion of the program is quite complex, but it is worth exploring, as many file-based applications will use similar code. I've made more details available in the bonus chapter "The RichBar Example" on the companion CD.

Besides file operations, the program supports copy and paste operations and font management. The copy and paste operations don't require an actual interaction with the clipboard, as the component can handle them with simple commands, such as:

```
RichEdit.CutToClipboard;
RichEdit.CopyToClipboard;
RichEdit.PasteFromClipboard;
RichEdit.Undo;
```

FIGURE 7.12:

The toolbar of the RichBar example. Notice the drop-down menu.


RichNote - C:\md6code\07\RichBar\Sample text.rtf
Times New Roman
clBlack
Small
Medium
Large
Sample Text
More text More text More text More text More text More text
1/1

It is a little more advanced to know when these operations (and the corresponding buttons) should be enabled. We can enable Copy and Cut buttons when some text is selected, in the OnSelectionChange event of the RichEdit control:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  tbtnCut.Enabled := RichEdit.SelLength > 0;
  tbtnCopy.Enabled := tbtnCut.Enabled;
end;
```

The Copy operation, instead, cannot be determined by an action of the user, as it depends on the content of the Clipboard, influenced also by other applications. One approach is to use a timer and check the clipboard content from time to time. A better approach is to use the OnIdle event of the Application object (or the ApplicationEvents component). As the

RichEdit supports multiple clipboard formats, the code cannot simply look at those, but should ask the component itself, using a low-level feature not surfaced by the Delphi control:

```
procedure TFormRichNote.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
begin
  // update toolbar buttons
  tbtnPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;
end;
```

Basic font management is given by the Bold and Italic buttons, which have similar code. The Bold button toggles the relative attribute from the selected text (or changes the style at the current edit position):

```
procedure TFormRichNote.BoldExecute(Sender: TObject);
begin
  with RichEdit.SelAttributes do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
end;
```

Again, the current status of the button is determined by the current selection, so we'll need to add the following line to the `RichEditSelectionChange` method:

```
tbtnBold.Down := fsBold in RichEdit.SelAttributes.Style;
```

A Menu and a Combo Box in a Toolbar

Besides a series of buttons, the RichBar example has a drop-down menu and a couple of combo boxes, a feature shared by many common applications. The drop-down button allows selection of the font size, while the combo boxes allow rapid selection of the font family and the font color. This second combo is actually built using a ColorBox control.

The Size button is connected to a PopupMenu component (called `SizeMenu`) using the `DropdownMenu` property. A user can press the button, firing its `OnClick` event as usual, or select the drop-down arrow, open the pop-up menu (see again Figure 7.12), and choose one of its options. This case has three possible font sizes, per the menu definition:

```
object SizeMenu: TPopupMenu
  object Small1: TMenuItem
    Tag = 10
    Caption = 'Small'
    OnClick = SetFontSize
  end
  object Medium1: TMenuItem
    Tag = 16
```

```
      Caption = 'Medium'
      OnClick = SetFontSize
    end
    object Large1: TMenuItem
      Tag = 32
      Caption = 'Large'
      OnClick = SetFontSize
    end
  end
```

Each menu item has a tag indicating the actual size of the font, activated by a shared event handler:

```
procedure TFormRichNote.SetFontSize(Sender: TObject);
begin
  RichEdit.SelAttributes.Size := (Sender as TMenuItem).Tag;
end;
```

As the ToolBar control is a full-featured control container, you can directly take an edit box, a combo box, and other controls and place them inside the toolbar. The combo box in the toolbar is initialized in the `FormCreate` method, which extracts the screen fonts available in the system:

```
ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (RichEdit.Font.Name)
```

The combo box initially displays the name of the default font used in the RichEdit control, which is set at design time. This value is recomputed each time the current selection changes, using the font of the selected text, along with the current color for the ColorBox:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ComboFont.ItemIndex :=
    ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name);
  ColorBox1.Selected := RichEdit.SelAttributes.Color;
end;
```

When a new font is selected from the combo box, the reverse action takes place. The text of the current combo box item is assigned as the name of the font for any text selected in the RichEdit control:

```
RichEdit.SelAttributes.Name := ComboFont.Text;
```

The selection of a color in the ColorBox activates similar code.

Toolbar Hints

Another common element in toolbars is the *fly-by hint*, also called *balloon help*—some text that briefly describes the button currently under the cursor. This text is usually displayed in a yellow box after the mouse cursor has remained steady over a button for a set amount of time. To add hints to an application's toolbar, simply set its ShowHints property to True and enter some text for the Hint property of each button (more on hints text in the next section, "A Simple Status Bar").

If you want to have more control on how hints are displayed, you can use some of the properties and events of the Application object. This global object has, among others, the following properties:

Property	Defines
HintColor	The background color of the hint window
HintPause	How long the cursor should remain on a component before hints are displayed
HintHidePause	How long the hint will be displayed
HintShortPause	How long the system should wait to display a hint if another hint has just been displayed

A program, for example, might allow a user to customize the hint background color by selecting a specific with the following code:

```
ColorDialog.Color := Application.HintColor;
if ColorDialog.Execute then
  Application.HintColor := ColorDialog.Color;
```

NOTE As an alternative, you can change the hint color by handling the OnShowHint property of the Application object. This handler can change the color of the hint just for specific controls. The OnShowHint event is used in the CustHint example described later in this chapter.

A Simple Status Bar

Building a status bar is even simpler than building a toolbar. Delphi includes a specific StatusBar component, based on the corresponding Windows common control (a similar control is available also in VisualCLX). This component can be used almost as a panel when its SimplePanel property is set to True. In this case, you can use the SimpleText property to output some text. The real advantage of this component, however, is that it allows you to define a number of subpanels just by activating the editor of its Panels property. (You can also display this property editor by double-clicking the status bar control.) Each subpanel has its own graphical attributes, which you can customize using the editor. Another feature of the status bar component is the "size grip" area added to the lower-right corner of the bar, which is useful for resizing the form itself. This is a typical element of the Windows user interface, and you can control it with the SizeGrip property.

There are various uses for a status bar. The most common is to display information about the menu item currently selected by the user. Besides this, a status bar often displays other information about the status of a program: the position of the cursor in a graphical application, the current line of text in a word processor, the status of the lock keys, the time and date, and so on. To show information on a panel, you simply use its `Text` property, generally using an expression like this:

```
StatusBar1.Panels[1].Text := 'message';
```

In the RichBar example, I've built a status bar with three panels, for command hints, the status of the Caps Lock key, and the current editing position. The StatusBar component of the example actually has four panels; we need to define the fourth in order to delimit the area of the third panel. The last panel, in fact, is always large enough to cover the remaining surface of the status bar.

TIP Again, for more detail on the RichBar program, see the bonus chapter "The RichBar Example" on the companion CD.

The panels are not independent components, so you cannot access them by name, only by position as in the preceding code snippet. A good solution to improve the readability of a program is to define a constant for each panel you want to use, and then use these constants when referring to the panels. This is my sample code:

```
const
  sbpMessage = 0;
  sbpCaps = 1;
  sbpPosition = 2;
```

In the first panel of the status bar, I want to display the hint message of the toolbar button. The program obtains this effect by handling the application's `OnHint` event, again using the ApplicationEvents component, and copying the current value of the application's `Hint` property to the status bar:

```
procedure TFormRichNote.ApplicationEvents1Hint (Sender: TObject);
begin
  StatusBar1.Panels[sbpMessage].Text := Application.Hint;
end;
```

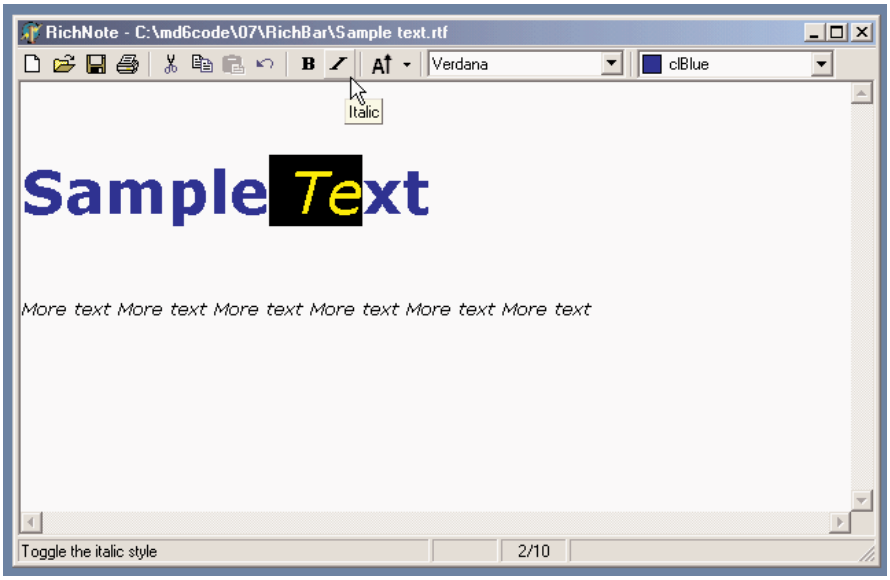
By default, this code displays in the status bar the same text of the fly-by hints. Actually, we can use the `Hint` property to specify different strings for the two cases, by writing a string divided into two portions by a separator, the pipe (`|`) character. For example, you might enter the following as the value of the `Hint` property:

```
'New|Create a new document'
```

The first portion of the string, *New*, is used by fly-by hints, and the second portion, *Create a new document*, by the status bar. You can see an example in Figure 7.13.

FIGURE 7.13:

The StatusBar of the RichBar example displays a more detailed description than the fly-by hint.


RichNote - C:\md6code\07\RichBar\Sample text.rtf

Sample Text

More text More text More text More text More text More text

Toggle the italic style

2/10

TIP

When the hint for a control is made up of two strings, you can use the `GetShortHint` and `GetLongHint` methods to extract the first (short) and second (long) substrings from the string you pass as a parameter, which is usually the value of the `Hint` property.

The second panel displays the status of the Caps Lock key, obtained by calling the `GetKeyState` API function, which returns a state number. If the low-order bit of this number is set (that is, if the number is odd), then the key is pressed. When do we check this state? I've decided to do this when the application is idle, so that this test is executed every time a key is pressed, but also as soon as a message reaches the window (in case the user changes this setting while working with another program). I've added to the `ApplicationEvents1Idle` handler a call to the custom `CheckCapslock` method, implemented as follows:

```
procedure TFormRichNote.CheckCapslock;
begin
  if Odd (GetKeyState (VK_CAPITAL)) then
    StatusBar1.Panels[sbpCaps].Text := 'CAPS'
  else
    StatusBar1.Panels[sbpCaps].Text := '';
end;
```

Finally, the program uses the third panel to display the current cursor position (measured in lines and characters per line) every time the selection changes. Because the `CaretPos` values are zero-based (that is, the upper-left corner is line 0, character 0), I've decided to add one to each value, to make them more reasonable for a casual user:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ...
  // update the position in the status bar
  StatusBar.Panels[sbpPosition].Text := Format ('%d/%d',
    [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```

Customizing the Hints

Just as we have added hints to an application's toolbar, we can add hints to forms or to the components of a form. For a large control, the hint will show up near the mouse cursor. In some cases, it is important to know that a program can freely customize how hints are displayed.

The simplest thing you can do is, change the value of the properties of the Application object as I mentioned at the end of the last section. To obtain more control over hints, you can customize them even further by assigning a method to the application's OnShowHint event. You need to either hook them up manually or—better—add an ApplicationEvents component to the form and handle its OnShowHint event.

The method you have to define has some interesting parameters, such as a string with the text of the hint, a Boolean flag for its activation, and a THintInfo structure with further information, including the control, the hint position, and its color. Each of the parameters is passed by reference, so you have a chance to change them and also modify the values of the THintInfo structure; for example, you can change the position of the hint window before it is displayed.

This is what I've done in the CustHint example, which shows the hint of the label at the center of its area. Here is what you can write to show the hint for the big label in the center of its surface:

```
procedure TForm1.ShowHint (var HintStr: string; var CanShow: Boolean;
  var HintInfo: THintInfo);
begin
  with HintInfo do
    if HintControl = Label1 then
      HintPos := HintControl.ClientToScreen (Point (
        HintControl.Width div 2, HintControl.Height div 2));
end;
```

The code has to retrieve the center of the generic control (the HintInfo.HintControl) and then convert its coordinates to screen coordinates, applying the ClientToScreen method to the control itself. We can further update the CustHint example in a different way. The RadioGroup control in the form has three radio buttons. However, these are not stand-alone components, but simply radio button clones painted on the surface of the radio group. What if we want to add a hint for each of them?

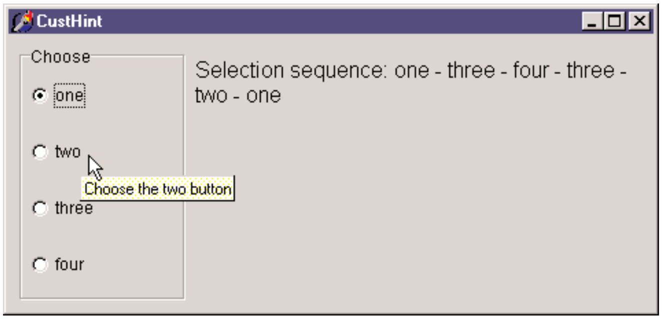
The CursorRect field of the THintInfo record can be used for this purpose. It indicates the area of the component that the cursor can move over without disabling the hint. When the cursor moves outside this area, Delphi hides the hint window. If we specify a different text for the hint and a different area for each of the radio buttons, we can in practice provide three different hints. Because computing the actual position of each radio button isn't easy, I've simply divided the surface of the radio group into as many equal parts as there are radio buttons. The text of the radio button (not the selected item, but the item under the cursor) is then added to the text of the hint:

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
var
  RadioItem, RadioHeight: Integer;
  RadioRect: TRect;
begin
  with HintInfo do
    if HintControl = Label1 ... // as before
    else
    if HintControl = RadioGroup1 then
    begin
      RadioHeight := (RadioGroup1.Height) div RadioGroup1.Items.Count;
      RadioItem := CursorPos.Y div RadioHeight;
      HintStr := 'Choose the ' + RadioGroup1.Items [RadioItem] + ' button';
      RadioRect := RadioGroup1.ClientRect;
      RadioRect.Top := RadioRect.Top + RadioHeight * RadioItem;
      RadioRect.Bottom := RadioRect.Top + RadioHeight;
      // assign the hints rect and pos
      CursorRect := RadioRect;
    end;
end;
```

The final part of the code builds the rectangle for the hint, starting with the rectangle corresponding to the client area of the component and moving its Top and Bottom values to the proper section of the RadioGroup1 component. The resulting effect is that each radio button of the radio group appears to have a specific hint, as shown in Figure 7.14.

FIGURE 7.14:

The RadioGroup control of the CustHint example shows a different hint, depending on which radio button the mouse is over.



Selection sequence: one - three - four - three - two - one

What's Next?

In this chapter I've discussed the use of some Delphi common controls, including the ListView, TreeView, PageControl, TabControl, ToolBar, StatusBar, and RichEdit. For each of these controls, I've built one example, trying to discuss it in the context of an actual application, even if most of the programs have been quite simple. I've also covered the Splitter component and various form-splitting techniques, the anchors for control positioning, and the customization of hints.

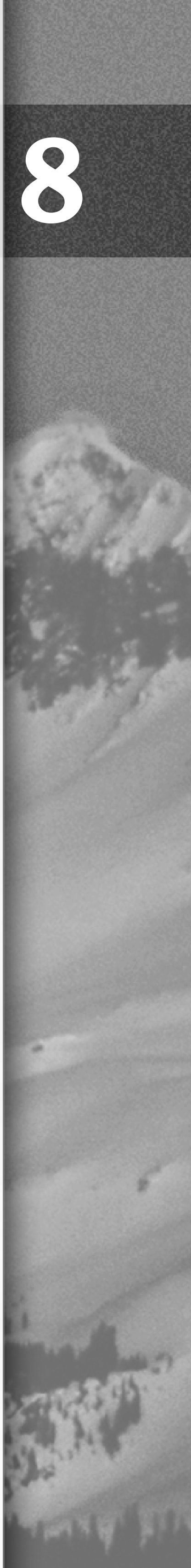
What is still missing is the development of an application with a complete user interface, including a menu and one or more toolbars. The reason I haven't covered this topic in the current chapter is that Delphi 6 adds quite a lot to VCL in this respect, including a complete architecture for letting the end users configure menus and toolbars based on a number of predefined actions. As this topic and related ones, such as docking toolbars, are complex, I've devoted the entire next chapter to them.

After this step, we'll move to the development of applications with multiple forms, including advanced dialog boxes, MDI, visual form inheritance, and the use of frames. All these topics are covered in Chapters 9 and 10.

CHAPTER 8

Building the User Interface

- Actions and ActionList
- Predefined actions in Delphi 6
- The ControlBar and CoolBar components
- Docking toolbars and other controls
- The Action Manager architecture



Modern Windows applications usually have multiple ways of giving a command, including menu items, toolbar buttons, shortcut menus, and so on. To separate the actual commands a user can give from their multiple representations in the user interface, Delphi has the idea of actions. In Delphi 6 this architecture has been largely extended to make the construction of the user interface on top of actions totally visual. You can now also easily let the user of your programs customize this interface, as happens in many professional programs.

This chapter focuses on actions, action lists and action managers, and the related components. It also covers a few related topics, such as toolbar container controls and toolbar docking, and docking in general.

The ActionList Component

Delphi's event architecture is very open: You can write a single event handler and connect it to the `OnClick` events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, as the event handler can use the `Sender` parameter to refer to the object that fired the event. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

To overcome this problem, Delphi 4 introduced an event-handling architecture based on actions. An *action* (or command) both indicates the operation to do when a menu item or button is clicked and determines the status of all the elements connected to the action. The connection of the action with the user interface of the linked controls is very important and should not be underestimated, because it is where you can get the real advantages of this architecture.

NOTE If you have ever written code using the MFC class library of Visual C++, you'll recognize that a Delphi action maps to both a command and a `CCommandUpdateUI` object. The Delphi architecture is more flexible, though, because it can be extended by subclassing the action classes.

There are many players in this event-handling architecture. The central role is certainly played by the action objects. An action object has a name, like any other component, and other properties that will be applied to the linked controls (called action clients). These properties include the `Caption`, the graphical representation (`ImageIndex`), the status (`Checked`, `Enabled`, and `Visible`), and the user feedback (`Hint` and `HelpContext`). There is also the `ShortCut` and a list of `SecondaryShortCuts`, the `AutoCheck` property for two-state actions, the help support, and a `Category` property used to arrange actions in logical groups.

The base class for an all action object is TBasicAction, which introduces the abstract core behavior of an action, without any specific binding or correction (not even to menu items or controls). The derived TContainedAction class introduces properties and methods that enable actions to appear in an action list or action manager. The further-derived TCustomAction class introduces support for the properties and methods of menu items and controls that are linked to action objects. Finally, there is the derived ready-to-use TAction class.

Each action object is connected to one or more client objects through an ActionLink object. Multiple controls, possibly of different types, can share the same action object, as indicated by their Action property. Technically, the ActionLink objects maintain a bidirectional connection between the client object and the action. The ActionLink object is required because the connection works in both directions. An operation on the object (such as a click) is forwarded to the action object and results in a call to its OnExecute event; an update to the status of the action object is reflected in the connected client controls. In other words, one or more client controls can create an ActionLink, which registers itself with the action object.

You should not set the properties of the client controls you connect with an action, because the action will override the property values of the client controls. For this reason, you should generally write the actions first and then create the menu items and buttons you want to connect with them. Note also that when an action has no OnExecute handler, the client control is automatically disabled (or grayed), unless the DisableIfNoHandler property is set to False.

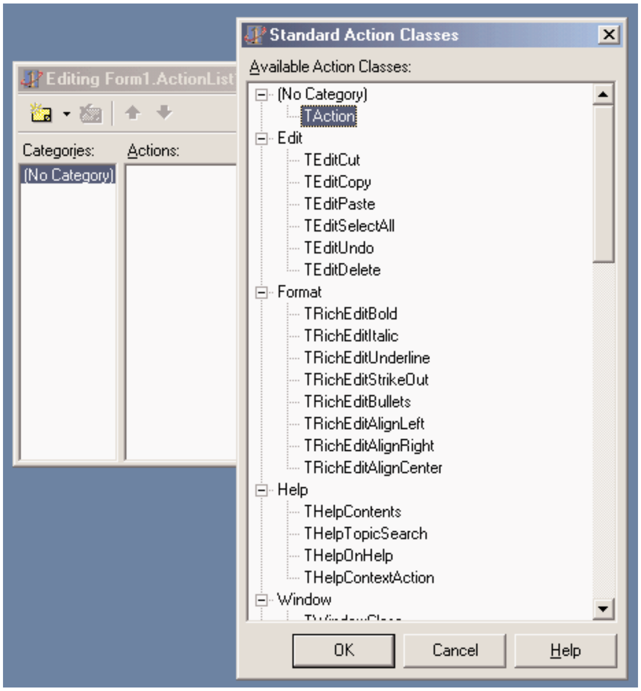
The client controls connected to actions are usually menu items and various types of buttons (push buttons, check boxes, radio buttons, speed buttons, toolbar buttons, and the like), but nothing prevents you from creating new components that hook into this architecture. Component writers can even define new actions, as we'll do in Chapter 11, and new link action objects.

Besides a client control, some actions can also have a target component. Some predefined actions hook to a specific target component (for examples, see the coverage of the DataSet components in the Chapter 13 section "Looking for Records in a Table"). Other actions automatically look for a target component in the form that supports the given action, starting with the active control.

Finally, the action objects are held by an ActionList component, the only class of the basic architecture that shows up on the Component Palette. The action list receives the execute actions that aren't handled by the specific action objects, firing the OnExecuteAction. If even the action list doesn't handle the action, Delphi calls the OnExecuteAction event of the Application object. The ActionList component has a special editor you can use to create several actions, as you can see in Figure 8.1.

FIGURE 8.1:

The ActionList component editor, with a list of predefined actions you can use



In the editor, actions are displayed in groups, as indicated by their `Category` property. By simply setting this property to a brand-new value, you instruct the editor to introduce a new category. These categories are basically logical groups, although in some cases a group of actions can work only on a specific type of target component. You might want to define a category for every pull-down menu or group them in some other logical way.

Predefined Actions in Delphi 6

With the action list editor, you can create a brand new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box, as shown in Figure 8.1. There are many predefined actions, which can be divided into logical groups:

File actions include open, save as, open with, run, print setup, and exit.

Edit actions are illustrated in the next example. They include cut, copy, paste, select all, undo, and delete.

RichEdit actions complement the edit actions for RichEdit controls and include bold, italic, underline, strikeout, bullets, and various alignment actions.

MDI window actions will be demonstrated in Chapter 10, as we examine the Multiple Document Interface approach. They include all the most common MDI operations: arrange, cascade, close, tile (horizontally or vertically), and minimize all.

Dataset actions relate to database tables and queries and will be discussed in Chapter 13. There are many dataset actions, representing all the main operations you can perform on a dataset.

Help actions allow you to activate the contents page or index of the Help file attached to the application.

Search actions include find, find first, find next, and replace.

Tab and Page control actions include previous page and next page navigation.

Dialog actions activate color, font, open, save, and print dialogs.

List actions include clear, copy, move, delete, and select all. These actions let you interact with a list control. Another group of actions, including static list, virtual list, and some support classes, allow the definition of lists that can be connected to a user interface. More on this topic is in the section "Using List Actions" toward the end of this chapter.

Web actions include browse URL, download URL, and send mail actions.

Tools actions include only the dialog to customize the action bars.

NOTE You can also define new custom actions and register them in Delphi's IDE, as we'll see in Chapter 11.

Besides handling the `OnExecute` event of the action and changing the status of the action to affect the user interface of the client controls, an action can also handle the `OnUpdate` event, which is activated when the application is idle. This gives you the opportunity to check the status of the application or the system and change the user interface of the controls accordingly. For example, the standard PasteEdit action enables the client controls only when there is some text in the Clipboard.

Actions in Practice

Now that you understand the main ideas behind this very important Delphi feature, let's try out an example from the companion CD. The program is called Actions and demonstrates a number of features of the action architecture. I began building it by placing a new ActionList component in its form and adding the three standard edit actions and a few custom ones. The form also has a panel with some speed buttons, a main menu, and a Memo control (the automatic target of the edit actions). Listing 8.1 is the list of the actions, extracted from the DFM file.

Listing 8.1: The actions of the Actions example

```
object ActionList1: TActionList
  Images = ImageList1
  object ActionCopy: TEditCopy
    Category = 'Edit'
    Caption = '&Copy'
    ShortCut = <Ctrl+C>
  end
  object ActionCut: TEditCut
    Category = 'Edit'
    Caption = 'Cu&t'
    ShortCut = <Ctrl+X>
  end
  object ActionPaste: TEditPaste
    Category = 'Edit'
    Caption = '&Paste'
    ShortCut = <Ctrl+V>
  end
  object ActionNew: TAction
    Category = 'File'
    Caption = '&New'
    ShortCut = <Ctrl+N>
    OnExecute = ActionNewExecute
  end
  object ActionExit: TAction
    Category = 'File'
    Caption = 'E&xit'
    ShortCut = <Alt+F4>
    OnExecute = ActionExitExecute
  end
  object NoAction: TAction
    Category = 'Test'
    Caption = '&No Action'
  end
  object ActionCount: TAction
    Category = 'Test'
    Caption = '&Count Chars'
    OnExecute = ActionCountExecute
    OnUpdate = ActionCountUpdate
  end
  object ActionBold: TAction
    Category = 'Edit'
    Caption = '&Bold'
    ShortCut = <Ctrl+B>
    OnExecute = ActionBoldExecute
  end
  object ActionEnable: TAction
    Category = 'Test'
    Caption = '&Enable NoAction'
```

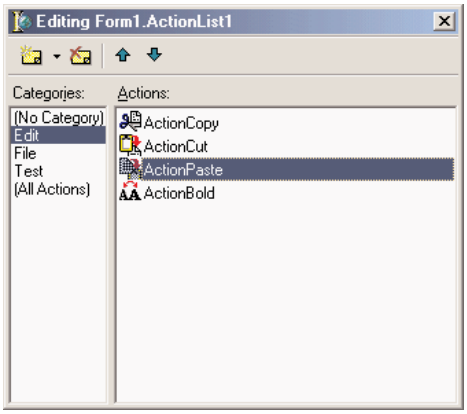
```
      OnExecute = ActionEnableExecute
    end
    object ActionSender: TAction
      Category = 'Test'
      Caption = 'Test &Sender'
      OnExecute = ActionSenderExecute
    end
  end
```

NOTE The shortcut keys are stored in the DFM files using virtual key numbers, which also include values for the Ctrl and Alt keys. In this and other listings throughout the book, I've replaced the numbers with the literal values, enclosing them in angle brackets.

All of these actions are connected to the items of a MainMenu component and some of them also to the buttons of a Toolbar control. Notice that the images selected in the ActionList control affect the actions in the editor only, as you can see in Figure 8.2. For the images of the ImageList to show up also in the menu items and in the toolbar buttons, you must also select the image list in the MainMenu and in the Toolbar components.

FIGURE 8.2:

The ActionList editor of the Actions example



The three predefined actions for the Edit menu don't have associated handlers, but these special objects have internal code to perform the related action on the active edit or memo control. These actions also enable and disable themselves, depending on the content of the Clipboard and on the existence of selected text in the active edit control. Most other actions have custom code, except for the `NoAction` object. Having no code, the menu item and the button connected with this command are disabled, even if the `Enabled` property of the action is set to True.

I’ve added to the example, and to the Test menu, another action that enables the menu item connected to the NoAction object:

```
procedure TForm1.ActionEnableExecute(Sender: TObject);
begin
  NoAction.DisableIfNoHandler := False;
  NoAction.Enabled := True;
  ActionEnable.Enabled := False;
end;
```

Simply setting Enabled to True will produce the effect for only a very short time, unless you set the DisableIfNoHandler property, as discussed in the previous section. Once this operation is done, I disable the current action, since there is no need to issue the same command again.

This is different from an action you can toggle, such as the Edit ➢ Bold menu item and the corresponding speed button. Here is the code of the Bold action:

```
procedure TForm1.ActionBoldExecute(Sender: TObject);
begin
  with Memo1.Font do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
  // toggle status
  ActionBold.Checked := not ActionBold.Checked;
end;
```

The ActionCount object has very simple code, but it demonstrates an OnUpdate handler; when the memo control is empty, it is automatically disabled. We could have obtained the same effect by handling the OnChange event of the memo control itself, but in general it might not always be possible or easy to determine the status of a control simply by handling one of its events. Here is the code of the two handlers of this action:

```
procedure TForm1.ActionCountExecute(Sender: TObject);
begin
  ShowMessage ('Characters: ' + IntToStr (Length (Memo1.Text)));
end;

procedure TForm1.ActionCountUpdate(Sender: TObject);
begin
  ActionCount.Enabled := Memo1.Text <> '';
end;
```

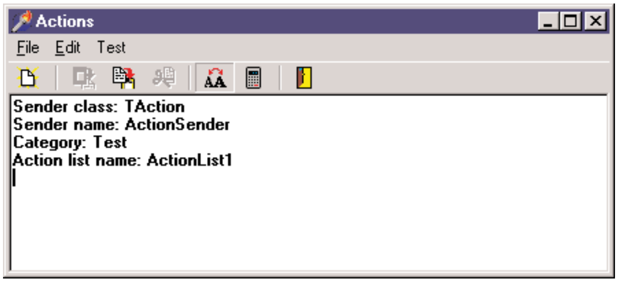
Finally, I’ve added a special action to test the sender object of the action event handler and get some other system information. Besides showing the object class and name, I’ve added code that accesses the action list object. I've done this mainly to show that you can access this information and how to do it:

```
procedure TForm1.ActionSenderExecute(Sender: TObject);
begin
  Memo1.Lines.Add ('Sender class: ' + Sender.ClassName);
  Memo1.Lines.Add ('Sender name: ' + (Sender as TComponent).Name);
  Memo1.Lines.Add ('Category: ' + (Sender as TAction).Category);
  Memo1.Lines.Add (
    'Action list name: ' + (Sender as TAction).ActionList.Name);
  end;
```

You can see the output of this code in Figure 8.3, along with the user interface of the example. Notice that the Sender is not the menu item you've selected, even if the event handler is connected to it. The Sender object, which fires the event, is the action, which intercepts the user operation.

FIGURE 8.3:

The Actions example, with a detailed description of the Sender of an Action object's OnExecute event



```
Sender class: TAction
Sender name: ActionSender
Category: Test
Action list name: ActionList1
```

Finally, keep in mind that you can also write handlers for the events of the ActionList object itself, which play the role of global handlers for all the actions of the list, and for the Application global object, which fires for all the actions of the application. Before calling the action's OnExecute event, in fact, Delphi activates the OnExecute event of the ActionList and the OnActionExecute event of the Application global object. These events can have a look at the action, eventually execute some shared code, and then stop the execution (using the Handled parameter) or let it reach the next level.

If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

NOTE When an action is executed, it searches for a control to play the role of the action target, by looking at the active control, the active form, and other controls on the form. For example, edit actions refer to the currently active control (if they inherit from TCustomEdit), while dataset controls look for the dataset connected with the data source of the data-aware control having the input focus. Other actions follow different approaches to find a target component, but the overall idea is shared by most standard actions.

The Toolbar and the ActionList of an Editor

In the previous chapter, I built the RichBar example to demonstrate the development of an editor with a toolbar and a status bar. Of course, I should have also added a menu bar to the form, but this would have created quite a few troubles in synchronizing the status of the toolbar buttons with those of the menu items. A very good solution to this problem is to use actions, which is what I've done in the MdEdit example, discussed in this section and available on the CD.

The application is based on an ActionList component, which includes actions for file handling and Clipboard support, with code similar to the RichBar version. The Font type and color selection is still based on combo boxes, so this doesn't involve action—same for the drop-down menu of the Size button. The menu, however, has a few extra commands, including one for character counting and one for changing the background color. These are based on actions, and the same happens for the three new paragraph justification buttons (and menu commands).

One of the key differences in this new version is that the code never refers to the status of the toolbar buttons, but eventually modifies the status of the actions. In other cases I've used the actions `OnUpdate` events. For example, the `RichEditSelectionChange` method doesn't update the status of the bold button, which is connected to an action with the following `OnUpdate` handler:

```
procedure TFormRichNote.acBoldUpdate(Sender: TObject);
begin
  acBold.Checked := fsBold in RichEdit.SelAttributes.Style;
end;
```

Similar `OnUpdate` event handlers are available for most actions, including the counting operations (available only if there is some text in the RichEdit control), the Save operation (available if the text has been modified), and the Cut and Copy operations (available only if some text is selected):

```
procedure TFormRichNote.acCountcharsUpdate(Sender: TObject);
begin
  acCountChars.Enabled := RichEdit.GetTextLen > 0;
end;

procedure TFormRichNote.acSaveUpdate(Sender: TObject);
begin
  acSave.Enabled := Modified;
end;

procedure TFormRichNote.acCutUpdate(Sender: TObject);
begin
  acCut.Enabled := RichEdit.SelLength > 0;
  acCopy.Enabled := acCut.Enabled;
end;
```

In the older example, the status of the Paste button was updated in the OnIdle event of the Application object. Now that we use actions we can convert it into yet another OnUpdate handler (see the preceding chapter for details on this code):

```
procedure TFormRichNote.acPasteUpdate(Sender: TObject);
begin
  acPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;
end;
```

Finally, the program has an addition compared to the last version: the three paragraph-alignment buttons. These toolbar buttons and the related menu items should work like radio buttons, being mutually exclusive with one of the three options always selected. For this reason the actions have the GroupIndex set to 1, the corresponding menu items have the RadioItem property set to True, and the three toolbar buttons have their Grouped property set to True and the AllowAllUp property set to False. (They are also visually enclosed between two separators.)

This is required so that the program can set the Checked property for the action corresponding to the current style, which avoids unchecking the other two actions directly. This code is part of the OnUpdate event of the action list, as it applies to multiple actions:

```
procedure TFormRichNote.ActionListUpdate(Action: TBasicAction;
  var Handled: Boolean);
begin
  // check the proper paragraph alignment
  case RichEdit.Paragraph.Alignment of
    taLeftJustify: acLeftAligned.Checked := True;
    taRightJustify: acRightAligned.Checked := True;
    taCenter: acCentered.Checked := True;
  end;
  // checks the caps lock status
  CheckCapslock;
end;
```

Finally, when one of these buttons is selected, the shared event handler uses the value of the Tag, set to the corresponding value of the TAlignment enumeration, to determine the proper alignment:

```
procedure TFormRichNote.ChangeAlignment(Sender: TObject);
begin
  RichEdit.Paragraph.Alignment := TAlignment ((Sender as TAction).Tag);
end;
```

Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement this differently. Delphi has two ready-to-use toolbar containers, the CoolBar and the ControlBar components. They have differences in their user interface, but the biggest one is that the CoolBar is a Win32 common control, part of the operating system, while the ControlBar is a VCL-based component.

Both components can host toolbar controls as well as some extra elements such as combo boxes and other controls. Actually, a toolbar can also replace the menu of an application, as we'll see later on.

We'll investigate the two components in the next two sections, but I want to emphasize here (without getting too far ahead of myself) that I generally favor the use of the ControlBar. It is based on VCL (and not subject to upgrade along with each minor release of Microsoft Internet Explorer), and its user interface is nicer and more similar to that of common office applications.

A Really Cool Toolbar

The CoolBar component is basically a collection of `TCoolBand` objects that you can activate by selecting the Band Editor item of the CoolBar shortcut menu, the `Bands` property, or the Object TreeView. You can customize the CoolBar component in many ways: You can set a bitmap for its background, add some bands to the `Bands` collection, and then assign to each band an existing component or component container. You can use any window-based control (not graphic controls), but only some of them will show up properly. If you want to have a bitmap on the background of the CoolBar, for example, you need to use partially transparent controls.

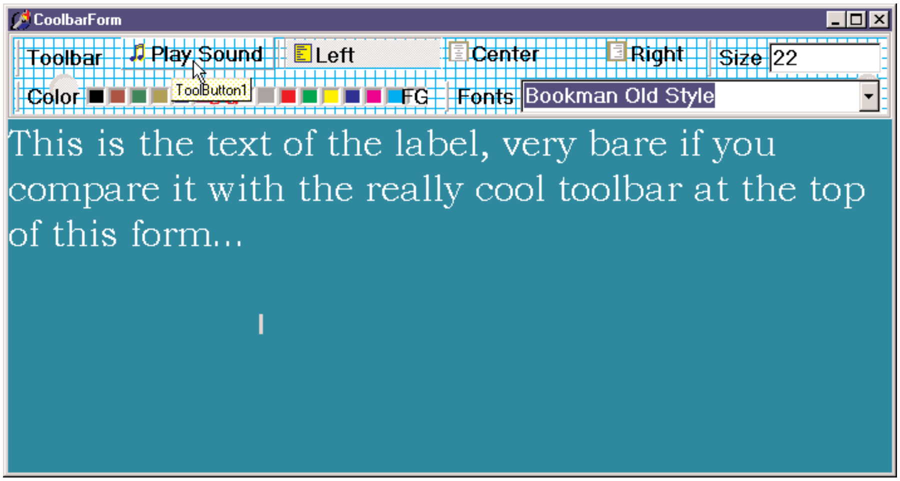
The typical component used in a CoolBar is the Toolbar (which can be made completely transparent), but combo boxes, edit boxes, and animation controls are also quite common. This is often inspired by the user interface of Internet Explorer, the first Microsoft application featuring the CoolBar component.

You can place one band on each line or all of them on the same line. Each would use a part of the available surface, and it would be automatically enlarged when the user clicks on its title. It is easier to use this new component than to explain it. Try it yourself or follow the description below, in which we build a new version of our continuing toolbar example based on a CoolBar control. You can see the form displayed by this application at run time in Figure 8.4.

The CoolBar example has a `TCoolBar` component with four bands, two for each of the two lines. The first band includes a subset of the toolbar of the previous example, this time adding an ImageList for the highlighted images. The second has an edit box used to set the font of the text; the third has a ColorGrid component, used to choose the font color and that of the background. The last band has a ComboBox control with the available fonts.

FIGURE 8.4:

The form of the CoolBar example at run time



This is the text of the label, very bare if you compare it with the really cool toolbar at the top of this form...

The user interface of the CoolBar component is really very nice, and Microsoft is increasingly using it in its applications. However, the Windows CoolBar control has had many different and incompatible versions, as Microsoft has released different versions of the common control library with different versions of the Internet Explorer. Some of these versions "broke" existing programs built with Delphi.

NOTE

It is interesting to note that Microsoft applications generally don't use the common control libraries. Word and Excel use their own internal versions of the common controls, and VB uses an OCX, not the common controls directly. Part of the reason that Borland had so much trouble with the common controls is that it uses them more (and in more ways) than even Microsoft does.

For this reason, Borland introduced (in Delphi 4) a toolbar container called the ControlBar. A control bar hosts several controls, as a CoolBar does, and offers a similar user interface that lets a user drag items and reorganize the toolbar at run time. A good example of the use of the ControlBar control is Delphi's own toolbar, but Microsoft applications use a very similar user interface.

The ControlBar

The ControlBar is a control container, and you build it just by placing other controls inside it, as you do with a panel (there is no list of `Bands` in it). Every control placed in the bar gets its own dragging area (a small panel with two vertical lines, on the left of the control), as you can see in Figure 8.5. For this reason, you should generally avoid placing specific buttons inside the ControlBar, but rather add containers with buttons inside them. Rather than using a panel, you should generally use one ToolBar control for every section of the ControlBar.

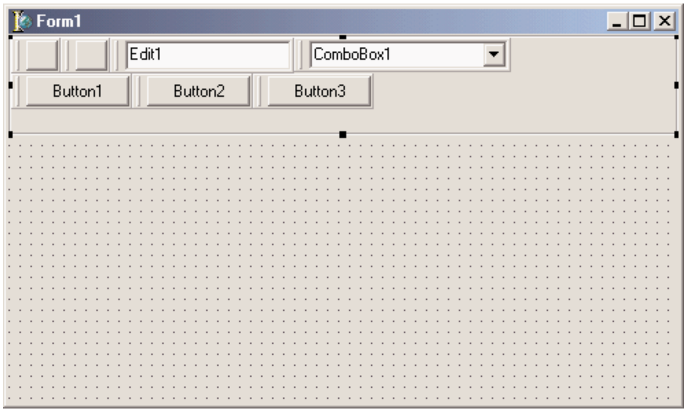


FIGURE 8.5: The ControlBar is a container that allows a user to drag all the elements, using the special drag bar on the side. Notice that each button gets a separate drag bar, something you'll generally try to avoid.

The MdEdit2 example is another version of the demo we've developed throughout the last and this chapter. I've basically grouped the buttons into three toolbars (instead of a single one) and left the two combo boxes as stand-alone controls. All these components are inside a ControlBar, so that a user can arrange them at runtime, as you can see in Figure 8.6.

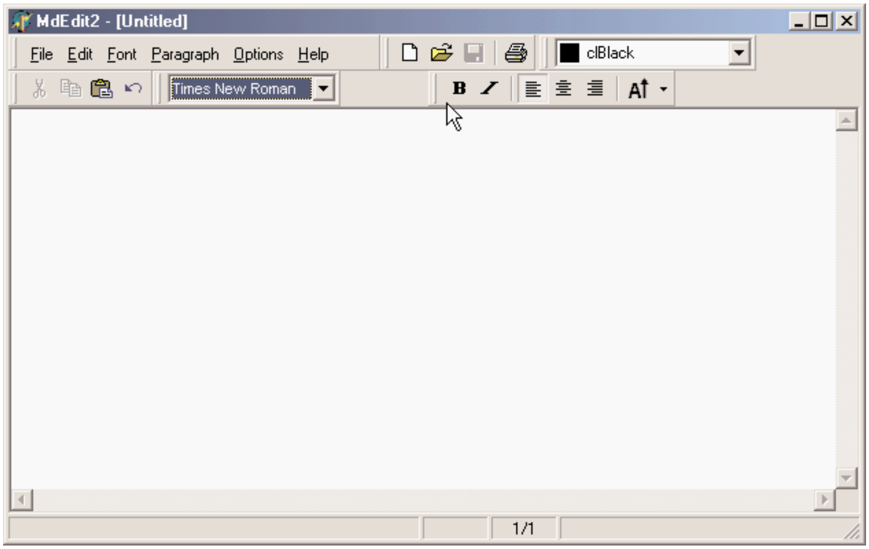


FIGURE 8.6: The MdEdit2 example at run time, while a user is rearranging the toolbars in the control bar

The following snippet of the DFM listing of the MdEdit2 example shows how the various toolbars and controls are embedded in the ControlBar component:

```
object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  PopupMenu = BarMenu
  object ToolBarFile: TToolBar
    Flat = True
    Images = Images
    Wrapable = False
    object ToolButton1: TToolButton
      Action = acNew
    end
    // more buttons...
  end
  object ToolBarEdit: TToolBar...
  object ToolBarFont: TToolBar...
  object ToolBarMenu: TToolBar
    AutoSize = True
    Flat = True
    Menu = MainMenu
  end
  object ComboFont: TComboBox
    Hint = 'Font Family'
    Style = csDropDownList
    OnClick = ComboFontClick
  end
  object ColorBox1: TColorBox...
end
```

To obtain the standard effect, you have to disable the edges of the toolbar controls and set their style to flat. Sizing all the controls alike, so that you obtain one or two rows of elements of the same height, is not as easy as it might seem at first. Some controls have automatic sizing or various constraints. In particular, to make the combo box the same height as the toolbars, you have to tweak the type and size of its font. Resizing the control itself has no effect.

The ControlBar also has a shortcut menu that allows you to show or hide each of the controls currently inside it. Instead of writing code specific to this example, I've implemented a more generic (and reusable) solution. The shortcut menu, called `BarMenu`, is empty at design time and is populated when the program starts:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
  mItem: TMenuItem;
```

```
begin
  ...
  // populate the control bar menu
  for I := 0 to ControlBar.ControlCount - 1 do
  begin
    mItem := TMenuItem.Create (Self);
    mItem.Caption := ControlBar.Controls [I].Name;
    mItem.Tag := Integer (ControlBar.Controls [I]);
    mItem.OnClick := BarMenuClick;
    BarMenu.Items.Add (mItem);
  end;
```

The BarMenuClick procedure is a single event handler that is used by all of the items of the menu and uses the Tag property of the Sender menu item to refer to the element of the ControlBar associated with the item in the FormCreate method:

```
procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
  aCtrl: TControl;
begin
  aCtrl := TControl ((Sender as TComponent).Tag);
  aCtrl.Visible := not aCtrl.Visible;
end;
```

Finally, the OnPopup event of the menu is used to refresh the check mark of the menu items:

```
procedure TFormRichNote.BarMenuPopup(Sender: TObject);
var
  I: Integer;
begin
  // update the menu checkmarks
  for I := 0 to BarMenu.Items.Count - 1 do
    BarMenu.Items [I].Checked := TControl (BarMenu.Items [I].Tag).Visible;
end;
```

A Menu in a Control Bar

If you look at the user interface of the MdEdit2 application, in Figure 8.6, you'll notice that the menu of the form actually shows up inside a toolbar, hosted by the control bar, and below the application caption. In prior versions of Delphi, this required writing some custom code. In Delphi 6, instead, all you have to do is to set the Menu property of the toolbar. You must also remove the main menu from the Menu property of the form, to avoid having two menus.

Delphi's Docking Support

Another feature added in Delphi 4 was support for *dockable* toolbars and controls. In other words, you can create a toolbar and move it to any of the sides of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

First of all, Delphi's docking support is connected with container controls, not with forms. A panel, a ControlBar, and other containers (technically, any control derived from `TWinControl`) can be set up as dock targets by enabling their `DockSite` property. You can also set the `AutoSize` property of these containers, so that they'll show up only if they actually hold a control.

To be able to drag a control (an object of any `TControl`-derived class) into the dock site, simply set its `DragKind` property to dkDock and its `DragMode` property to dmAutomatic. This way, the control can be dragged away from its current position into a new docking container. To undock a component and move it to a special form, you can set its `FloatingDockSiteClass` property to `TCustomDockForm` (to use a predefined stand-alone form with a small caption).

All the docking and undocking operations can be tracked by using special events of the component being dragged (`OnStartDock` and `OnEndDock`) and the component that will receive the docked control (`OnDragOver` and `OnDragDrop`). These docking events are very similar to the dragging events available in earlier versions of Delphi.

There are also commands you can use to accomplish docking operations in code and to explore the status of a docking container. Every control can be moved to a different location using the `Dock`, `ManualDock`, and `ManualFloat` methods. A container has a `DockClientCount` property, indicating the number of docked controls, and a `DockClients` property, with the array of these controls.

Moreover, if the dock container has the `UseDockManager` property set to True, you'll be able to use the `DockManager` property, which implements the `IDockManager` interface. This interface has many features you can use to customize the behavior of a dock container, even including support for streaming its status.

As you can see from this brief description, docking support in Delphi is based on a large number of properties, events, methods and objects (such as dock zones and dock trees)—more features than we have room to explore in detail. The next example introduces the main features you'll generally need.

NOTE Docking support in not currently available in VisualCLX on either platform.

Docking Toolbars in ControlBars

In the MdEdit2 example, already discussed, I've included docking support. The program has a second ControlBar at the bottom of the form, which accepts dragging one of the toolbars in the ControlBar at the top. Since both toolbar containers have the AutoSize property set to True, they are automatically removed when the host contains no controls. I've also set to True the AutoDrag and AutoDock properties of both ControlBars.

Actually, I had to place the bottom ControlBar inside a panel, together with the RichEdit control. Without this trick, the ControlBar, when activated and automatically resized, kept moving below the status bar, which I don't think is the correct behavior. Because, in the example, the ControlBar is the only control of the panel aligned to the bottom, there is no possible confusion.

To let users drag the toolbars out of the original container, all you have to do is, once again (as stated previously), set their DragKind property to dkDock and their DragMode property to dmAutomatic. The only two exceptions are the menu toolbar, which I decided to keep close to the typical position of a menu bar, and the ColorBox control, as unlike the combo box this component doesn't expose the DragMode and DragKind properties. (Actually, in the FormCreate method of the example, you'll find code you can use to activate docking for the component, based on the "protected hack" discussed in Chapter 3.) The Fonts combo box can be dragged, but I don't want to let a user dock it in the lower control bar. To implement this constraint, I've used the control bar's OnDockOver event handler, by accepting the docking operation only for toolbars:

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept := Source.Control is TToolbar;
end;
```

When you move one of the toolbars outside of any container, Delphi automatically creates a floating form; you might be tempted to set it back by closing the floating form. This doesn't work, as the floating form is removed along with the toolbar it contains. However, you can use the shortcut menu of the topmost ControlBar, attached also to the other ControlBar, to show this hidden toolbar.

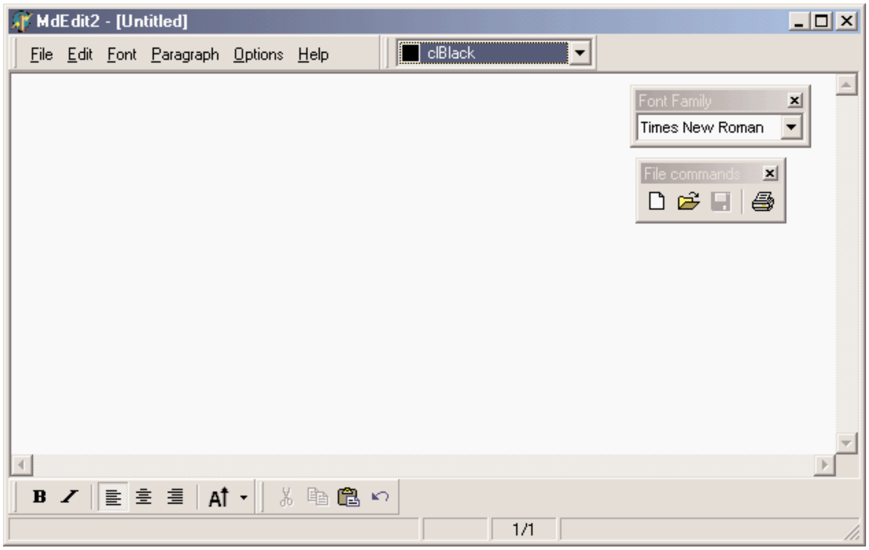
The floating form created by Delphi to host undocked controls has a thin caption, the so-called *toolbar caption*, which by default has no text. For this reason, I've added some code to the OnEndDock event of each dockable control, to set the caption of the newly created form into which the control is docked. To avoid a custom data structure for this information, I've used the text of the Hint property of these controls, which is basically not used, to provide a suitable caption:

```
procedure TFormRichNote.EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  if Target is TCustomForm then
    TCustomForm(Target).Caption := GetShortHint((Sender as TControl).Hint);
end;
```

You can see an example of this effect in the MdEdit2 program in Figure 8.7. Another extension of the example, one which I haven't done, could be the addition of dock areas on the two sides of the form. The only extra effort this requires would be a routine to turn the toolbars vertically, instead of horizontally. This basically implies switching the Left and Top properties of each button, after disabling the automatic sizing.

FIGURE 8.7:

The MdEdit2 example allows you to dock the toolbars (but not the menu) at the top or bottom of the form or to leave them floating.



Controlling Docking Operations

Delphi provides many events and methods that give you a lot of control over docking operations, including a dock manager. To explore some of these features, try out the DockTest example, a test bed for docking operations. The program assigns the FloatingDockSiteClass property of a Memo component to TForm2, so that you can design specific features and add them to the floating frame that will host the control when it is floating, instead of using an instance of the default TCustomDockForm class.

Another feature of the program is that it handles the OnDockOver and OnDockDrop events of a dock host panel to display messages to the user, such as the number of controls currently docked:

```
procedure TForm1.Panel1DockDrop(Sender: TObject; Source: TDragDockObject;
  X, Y: Integer);
begin
  Caption := 'Docked: ' + IntToStr (Panel1.DockClientCount);
end;
```

In the same way, the program also handles the main form's docking events. Another control, a list box, has a shortcut menu you can invoke to perform docking and undocking operations in code, without the usual mouse dragging:

```
procedure TForm1.DocktoPanel1Click(Sender: TObject);
begin
  // dock to the panel
  ListBox1.ManualDock (Panel1, Panel1, alBottom);
end;

procedure TForm1.DocktoForm1Click(Sender: TObject);
begin
  // dock to the current form
  ListBox1.Dock (Self, Rect (200, 100, 100, 100));
end;

procedure TForm1.Floating1Click(Sender: TObject);
begin
  // toggle the floating status
  if ListBox1.Floating then
    ListBox1.ManualDock (Panel1, Panel1, alBottom)
  else
    ListBox1.ManualFloat (Rect (100, 100, 200, 300));
  Floating1.Checked := ListBox1.Floating;
end;
```

The final feature of the example is probably the most interesting one: Every time the program closes, it saves the current docking status of the panel, using the dock manager support. When the program is reopened, it reapplies the docking information, restoring the previous configuration of the windows. The program does this only with the panel, so the other floating windows will be displayed in their original positions. Here is the code for saving and loading:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  FileStr: TFileStream;
begin
```

```
    if Panel1.DockClientCount > 0 then
    begin
      FileStr := TFileStream.Create (DockFileName, fmCreate or fmOpenWrite);
      try
        Panel1.DockManager.SaveToStream (FileStr);
      finally
        FileStr.Free;
      end;
    end
    else
      // remove the file
      DeleteFile (DockFileName);
  end;

  procedure TForm1.FormCreate(Sender: TObject);
  var
    FileStr: TFileStream;
  begin
    // reload the settings
    DockFileName := ExtractFilePath (Application.Exename) + 'dock.dck';
    if FileExists (DockFileName) then
    begin
      FileStr := TFileStream.Create (DockFileName, fmOpenRead);
      try
        Panel1.DockManager.LoadFromStream (FileStr);
      finally
        FileStr.Free;
      end;
    end;
    Panel1.DockManager.ResetBounds (True);
  end;
```

There are more features one might theoretically add to a docking program, but to add those you should remove other features, as some of them might conflict. For example, automatic alignments don't work terribly well with the docking manager's code for restoring. I suggest you take this program and explore its behavior, extending it to support the type of user interface you prefer.

NOTE Remember that although docking panels make an application look nice, some users get confused by the fact that their toolbars might disappear or be in a different position than they are used to. Don't overuse the docking features, or some of your inexperienced users may get lost.

Docking to a PageControl

Another interesting feature of page controls is the specific support for docking. As you dock a new control over a PageControl, a new page is automatically added to host it, as you can easily see in the Delphi environment. To accomplish this, you simply set the PageControl as a dock host and activate docking for the client controls. This works best when you have secondary forms you want to host. Moreover, if you want to be able to move the entire PageControl into a floating window and then dock it back, you'll need a docking panel in the main form.

This is exactly what I've done in the DockPage example, which has a main form with the following settings:

```
object Form1: TForm1
  Caption = 'Docking Pages'
  object Panel1: TPanel
    Align = alLeft
    DockSite = True
    OnMouseDown = Panel1MouseDown
    object PageControl1: TPageControl
      ActivePage = TabSheet1
      Align = alClient
      DockSite = True
      DragKind = dkDock
      object TabSheet1: TTabSheet
        Caption = 'List'
        object ListBox1: TListBox
          Align = alClient
        end
      end
    end
  end
  object Splitter1: TSplitter
    Cursor = crHSplit
  end
  object Memo1: TMemo
    Align = alClient
  end
end
```

Notice that the Panel has the `UseDockManager` property set to True and that the PageControl invariably hosts a page with a list box, as when you remove all of the pages, the code used for automatic sizing of dock containers might cause you some trouble. Now the program has two other forms, with similar settings (although they host different controls):

```
object Form2: TForm2
  Caption = 'Small Editor'
  DragKind = dkDock
  DragMode = dmAutomatic
  object Memo1: TMemo
    Align = alClient
  end
end
```

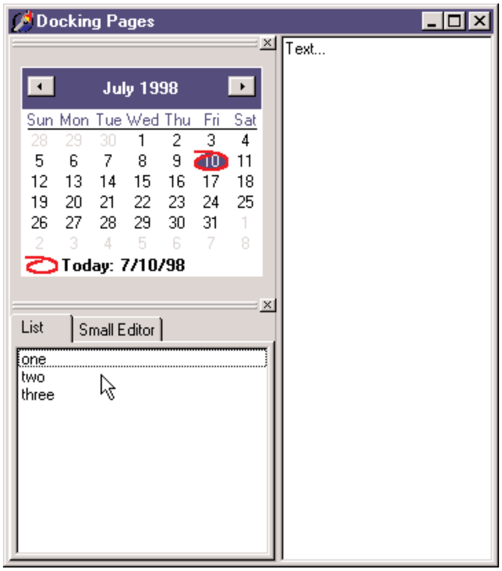
You can drag these forms onto the page control to add new pages to it, with captions corresponding with the form titles. You can also undock each of these controls and even the entire PageControl. To do this, the program doesn't enable automatic dragging, which would make it impossible to switch pages anymore. Instead, the feature is activated when the user clicks on the area of the PageControl that has no tabs—that is, on the underlying panel:

```
procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  PageControl1.BeginDrag (False, 10);
end;
```

You can test this behavior by running the DockPage example, although Figure 8.8 tries to depict it. Notice that when you remove the PageControl from the main form, you can directly dock the other forms to the panel and then split the area with other controls. This is the situation captured by the figure.

FIGURE 8.8:

The main form of the DockPage example after a form has been docked to the page control on the left. Notice that another form uses part of the area of a hosting panel.



Text...

Sun	Mon	Tue	Wed	Thu	Fri	Sat
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

Today: 7/10/98

one
two
three

The ActionManager Architecture

We have seen that actions and the ActionManager component can play a central role in the development of Delphi applications, since they allow a much better separation of the user interface from the actual code of the application. The user interface, in fact, can now easily change without impacting the code too much. The drawback of this approach is that a programmer has more work to do. To have a new menu item, you need to add the corresponding action first, than move to the menu, add the menu item, and connect it to the action.

To solve this issue, and to provider developers and end users with some advanced features, Delphi 6 introduces a brand new architecture, based on the ActionManager component, which largely extends the role of actions. The ActionManager, in fact, has a collection of actions but also a collection of toolbars and menus tied to them. The development of these toolbars and menus is completely visual: you drag actions from a special component editor of the ActionManager to the toolbars to have the buttons you need. Moreover, you can let the end user of your programs do the same operation, and rearrange their own toolbars and menus starting with the actions you provide them.

In other words, using this architecture allows you to build applications with a modern user interface, customizable by the user. The menu can show only the recently used items (as many Microsoft programs do, nowadays), allows for animation, and more.

This architecture is centered on the ActionManager component, but includes also a few others components found at the end of the Additional page of the palette:

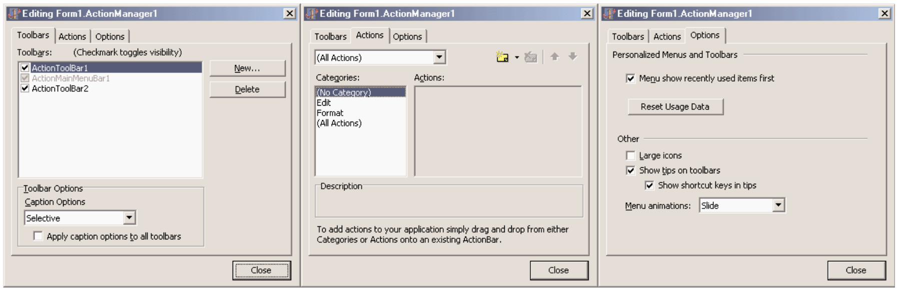
- The ActionManager component is a replacement of the ActionList (but can also use one or more existing ActionLists) adding to the architecture visual containers of actions.
- The ActionMainMenuBar control is a toolbar used to display the menu of an application based on the actions of an ActionManager component.
- The ActionToolBar control is a toolbar used to host buttons based on the actions of an ActionManager component.
- The CustomizeDlg component includes the dialog box you can use to let users customize the user interface of an application based on the ActionManager component.

Building a Simple Demo

As this architecture is mostly a visual architecture, a demo is probably worth more than a general discussion (although a printed book is not the best way to discuss a highly visual series of operations). To create a sample program based on this architecture, first drop an ActionManager component on a form, then double click it to open its component editor, shown in Figure 8.9. Notice that this editor is not modal, so you can keep it open while doing other operations in Delphi. Consider also that this same dialog box is displayed by the CustomizeDlg component, although with some limited features (for example, adding new actions is disabled).

FIGURE 8.9:

The three pages of the ActionManager editor dialog box



Toolbars: (Checkmark toggles visibility)

☑ ActionToolBar1
☑ ActionMainMenuBar1
☑ ActionToolBar2

Caption Options: Selective

□ Apply caption options to all toolbars

(All Actions)

Categories: (No Category) Edit Format (All Actions)

Actions:

To add actions to your application simply drag and drop from either Categories or Actions onto an existing ActionBar.

Personalized Menus and Toolbars

☑ Menu show recently used items first

□ Large icons
☑ Show tips on toolbars
☑ Show shortcut keys in tips

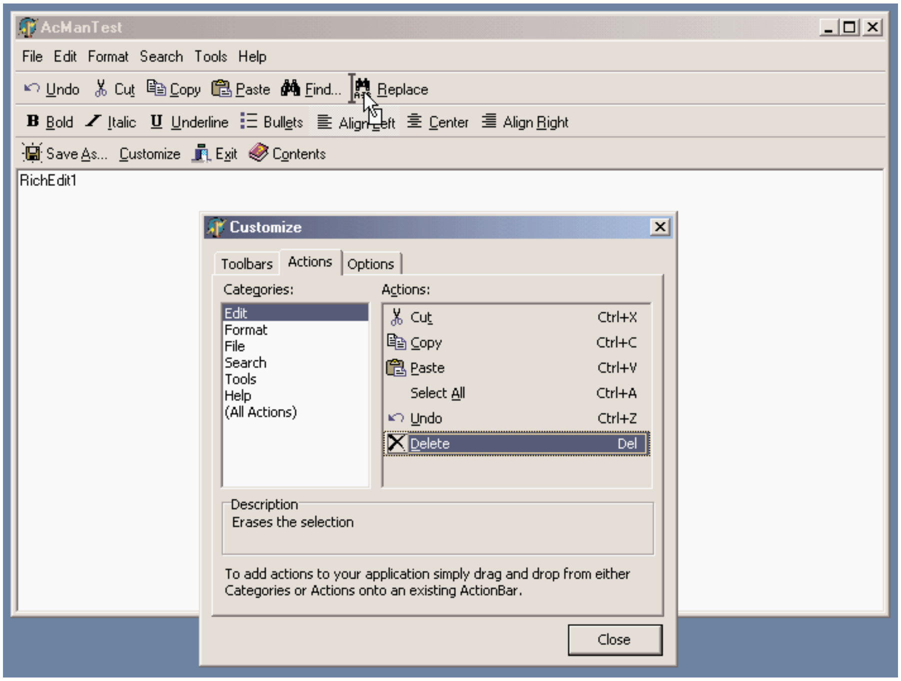
Menu animations: Slide

- The first page of this editor provides a list of visual containers of actions (toolbars or menus). You add new toolbars by clicking the New button. To add new menus, you have to add the corresponding component to the form, then open the `ActionBars` collection of the ActionManager, select an action bar or add a new one, and hook the menu to it using the `ActionBar` property. These are the same steps you could follow to connect a new toolbar to this architecture at run time.
- The second page of the ActionManager editor is very similar to the ActionList editor, providing a way to add new standard or custom action, arrange them in categories, and change their order. The new feature of this page, though, is that fact you can drag a category or a single action from it and drop it onto an action bar control. If you drag a category to a menu, you obtain a pull-down menu with all of the items of the category; if you drag it to a toolbar, each of the actions of the category gets a button on the toolbar. If you drag a single action to a toolbar, you get the corresponding button; if you drag it to the menu, you get a direct menu command, which is something you should generally avoid.
- The last page of the ActionManager editor allows you (and optionally an end user) to activate the display of recently used menu items and to modify some of the visual properties of the toolbars.

The AcManTest program is an example that uses some of the standard actions and a RichEdit control to showcase the use of this architecture (I haven't actually written any custom code to make the actions work better, as I wanted to focus only on the action manager for this example). You can experiment with it at design time or run it, click the Customize button, and see what an end user can do to customize the application (see Figure 8.10).

FIGURE 8.10:

Using the CustomizeDlg component, you can let a user customize the toolbars and the menu of an application, simply by dragging items from the dialog box or moving them around in the actions bars themselves.



Erases the selection

To add actions to your application simply drag and drop from either Categories or Actions onto an existing ActionBar.

Actually, in the program you can prevent the user from doing some operations on actions. Any specific element of the user interface (a `TActionClient` object) has a `ChangedAllowed` property that you can use to disable modify, move, and delete operations. Any action client container (the visual bars) has a property to disable hiding itself (`AllowHiding` by default is set to True). Each ActionBar `Items` collection has a `Customizable` option you can turn off to disable all user changes to the entire bar.

TIP

When I say "ActionBar" I don't mean the visual toolbars containing action items, but the items of the `ActionBars` collection of the ActionManager component, which in turn has an `Items` collection. The best way to understand this structure is to look at the sub-tree displayed by the Object TreeView for an ActionManager component. Each `TActionBar` collection item has an actual `TCustomActionBar` visual component connected, but not the reverse (so, for example, you cannot reach this `Customizable` property if you start by selecting the visual toolbar). Due to the similarity of the two names, it can take a while to understand what the Delphi help actually means.

To make user settings persistent, I've connected a file (called `settings`) to the `FileName` property of the ActionManager component. When you assign this property, you should enter a name of the file you want to use; when you start the program, the file will be created for you by the ActionManager.

The persistency is accomplished by streaming each ActionClientItem connected with the action manager. As these action client items are based on the user settings and maintain state information, a single file collects both user changes to the interface and usage data.

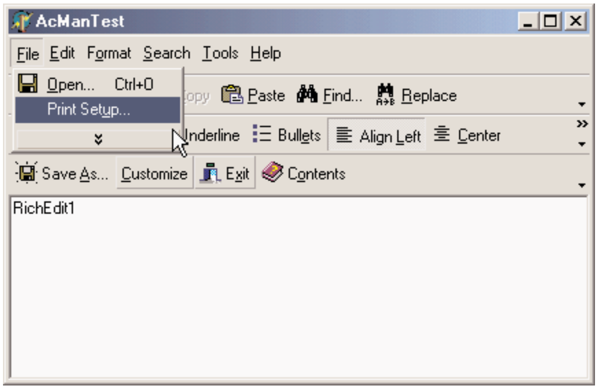
Since Delphi stores user setting and status information in a file you provide, you can make your application support multiple users on a single computer. Simply use a file of settings for each of them and connect it to the action manager as the program starts (using the current user of the computer or after some custom login). Another possibility is to store these settings over the network, so that even when a user moves to a different computer, the current personal settings will move along.

Least-Recently Used Menu Items

Once a file for the user settings is available, the ActionManager will save into it the user preferences and also use it to track the user activity. This is essential to let the system remove menu items which haven't been used for some time, making them available in an extended menu, using the same user interface adopted by Microsoft (see Figure 8.11 for an actual example).

FIGURE 8.11:

The ActionManager disables least recently used menu items that you can still see by selecting the menu extension command.



The ActionManager doesn't simply show the least recently used items: it allows you to customize this behavior in a very precise way. Each action bar has a `SessionCount` property that keeps track of the number of times the application has been executed. Each ActionClientItem has a `LastSession` property and a `UsageCount` property used to track user operations. Notice, by the way, that a user can reset all this dynamic information by using the Reset Usage Data button of the customization dialog.

The system calculates the number of sessions the action has gone unused, by computing the difference between the number of times the application has been executed (SessionCount) and the last session in which the action has been used (LastSession). The value of UsageCount is used to look up in the PrioritySchedule how many sessions the items can go unused before it is removed. In other words, the PrioritySchedule maps each the usage count with a number of *unused* sessions. By modifying the PrioritySchedule, you can determine how fast the items are removed in case they are not used.

You can also prevent this system to be activated for specific actions or groups of actions. The Items property of the ActionBars of the ActionManager has a HideUnused property you can toggle to disable this feature for an entire menu. To make a specific item always visible, regardless of the actual usage, you can also set its UsageCount property to –1. However, the user settings might override this value.

To understand a little better how this system works, I've added a custom action (ActionShowStatus) to the AcManTest example. The action has the following code that saves the current action manager settings to a memory stream, converts it to text, and shows it inside the memo (refer to Chapter 5 for more information about streaming):

```
procedure TForm1.ActionShowStatusExecute(Sender: TObject);
var
  memStr, memStr2: TMemoryStream;
begin
  memStr := TMemoryStream.Create;
  try
    memStr2 := TMemoryStream.Create;
    try
      ActionManager1.SaveToStream(memStr);
      memStr.Position := 0;
      ObjectBinaryToText(memStr, memStr2);
      memStr2.Position := 0;
      RichEdit1.Lines.LoadFromStream(memStr2);
    finally
      memStr2.Free;
    end;
  finally
    memStr.Free;
  end;
end;
```

The output you obtain is the textual version of the settings file automatically updated at each execution of the program. Here a small portion of this file, with the details of one of pull-down menus and plenty of extra comments:

```
item // File pulldown of the main menu action bar
  Items = <
```

```
      item
        Action = Form1.FileOpen1
        LastSession = 19 // was used in the last session
        UsageCount = 4 // was used four times
      end
      item
        Action = Form1.FileSaveAs1 // never used
      end
      item
        Action = Form1.FilePrintSetup1
        LastSession = 7 // used some time ago
        UsageCount = 1 // only once
      end
      item
        Action = Form1.FileRun1 // never used
      end
      item
        Action = Form1.FileExit1 // never used
      end>
    Caption = '&File'
    LastSession = 19
    UsageCount = 5 // the sum of the usage count of the items
  end
```

Porting an Existing Program

If this architecture is nice, you'll probably need to redo most of your applications to take advantage of it. However, if you're already using actions (with the ActionList component), this conversion will be much simpler. In fact, the ActionManager has its own set of actions but can also use actions from another ActionManager or ActionList. The `LinkedActionLists` property of the ActionManager is a collection of other containers of actions (ActionLists or ActionManagers), which can be associated with the current one. Associating all the various groups of action is useful to let a user customize the entire user interface with a single dialog box.

If you hook external actions and open the ActionManager editor, you'll see in the Actions page a combo box listing the current ActionManager plus the other action containers linked to it. You can choose one of these containers to see its set of actions and change their properties. The All Action option of this combo box allows you to work on all of the actions from the various containers at once, but I've noticed that at startup it is selected but not always *effective*. Reselect it to actually see all of the actions.

As an example of porting an existing application, I've extended the program built throughout this chapter, into the MdEdit3 example. This example uses the same action list of the previous version hooked to an ActionManager that has the extra customize property, to let users rearrange the user interface. Differently from the earlier AcManDemo program, the MdEdit3 example uses a ControlBar as a container for the action bars (a menu, three toolbars, and the usual combo boxes) and has full support for dragging them outside of the container as floating bars and dropping them into the lower ControlBar.

To accomplish this, I only had to modify the source code slightly to refer to the new classes for the containers (that is, TCustomActionToolBar instead of TToolBar) in the ControlBar-LowerDockOver method. I also found out that the OnEndDock event of the ActionToolBar component passes as parameter an empty target when the system creates a floating form to host the control, so that I couldn't easily give to this forms a new custom caption (see the EndDock method of the form).

Using List Actions

We'll see more examples of the use of this architecture in the chapters devoted to MDI and database programming. For the moment, I just want to add an extra example showing how to use a rather complex group of standard actions introduced in Delphi 6, the list actions. List actions, in fact, comprise two different groups. Some of them (such as the Move, Copy, Delete, Clear, and Select All) actions are normal actions working on list boxes or other lists. The VirtualListAction and StaticListAction elements, instead, define actions based multiple choices, which are going to be displayed in a toolbar as a combo box.

The ListActions demo highlights both groups of list actions, as its ActionManager has five of them, displayed on two separate toolbars. This is a summary of the actions of the actions manager (I've omitted the action bars portion of the component's DFM file):

```
object ActionManager1: TActionManager
  ActionBars.SessionCount = 1
  ActionBars = <...>
  object StaticListAction1: TStaticListAction
    Caption = 'Numbers'
    Items.CaseSensitive = False
    Items.SortType = stNone
    Items = <
      item
        Caption = 'one'
      end
      item
        Caption = 'two'
      end
      ...>
    OnItemSelected = ListActionItemSelected
  end
  object VirtualListAction1: TVirtualListAction
    Caption = 'Items'
```

```
      OnGetItem = VirtualListAction1GetItem
      OnGetItemCount = VirtualListAction1GetItemCount
      OnItemSelected = ListActionItemSelected
    end
    object ListControlCopySelection1: TListControlCopySelection
      Caption = 'Copy'
      Destination = ListBox2
      ListControl = ListBox1
    end
    object ListControlDeleteSelection1: TListControlDeleteSelection
      Caption = 'Delete'
    end
    object ListControlMoveSelection2: TListControlMoveSelection
      Caption = 'Move'
      Destination = ListBox2
      ListControl = ListBox1
    end
  end
```

The program has also two list boxes in its form, used as action targets. The Copy and Move actions are tied to these two list boxes by their `ListControl` and `Destination` properties. The Delete action, instead, automatically works with the list box having the input focus.

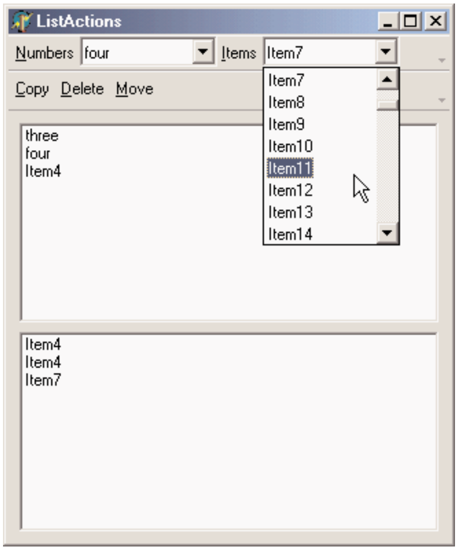
The StaticListAction defines a series of alternative items, in its `Items` collection. This is not a plain string list, as any item has also an `ImageIndex`, which allows turning the combo box in graphical selection. You can, of course, add more items to this list programmatically. However, in case of a highly dynamic list, you can also use the VirtualListAction. This component doesn't define a list of items but has two events you can use to provide strings and images for the list. The `OnGetItemCount` event allows you to indicate the number of items to display; the `OnGetItem` event is then called for each specific item.

In the ListActions demo, the VirtualListAction has the following event handlers for its definition, producing the list you can see in the active combo box of Figure 8.12:

```
procedure TForm1.VirtualListAction1GetItemCount(Sender: TCustomListAction;
  var Count: Integer);
begin
  Count := 100;
end;

procedure TForm1.VirtualListAction1GetItem(Sender: TCustomListAction;
  const Index: Integer; var Value: String;
  var ImageIndex: Integer; var Data: Pointer);
begin
  Value := 'Item' + IntToStr (Index);
end;
```

FIGURE 8.12:

The ListActions application has a toolbar hosting a static list and a virtual one.



NOTE I thought that the virtual action items were actually requested only when needed to display them, making this actually a virtual list. Instead, all the items are created right away, as you can prove by enabling the commented code of the `VirtualListAction1GetItem` method (not in the listing above), which adds to each item the time its string is requested.

Both the static and the virtual list have an `OnItemSelected` event. In the shared event handler, I've written the following code, to add the current item to the first list box of the form:

```
procedure TForm1.ListActionItemSelected(Sender: TCustomListAction;
  Control: TControl);
begin
  ListBox1.Items.Add ((Control as TCustomActionCombo).SelText);
end;
```

In this case, the sender is the custom action list, but the `ItemIndex` property of this list is not updated with the selected item. However, accessing the visual control that displays the list, we can obtain the value of the selected item.

What's Next?

In this chapter, I've introduced the use of actions, the actions list, and action manager architectures. As you've seen, this is an extremely powerful architecture to separate the user interface from the actual code of your applications, which uses and refers to the actions and not the menu items or toolbar button related to them. The Delphi 6 extension of this architecture allows users of your programs to have a lot of control, and makes your applications resemble high-end programs without much effort on your part. The same architecture is also very handy to let you design the user interface of your program, regardless of whether you give this ability to users.

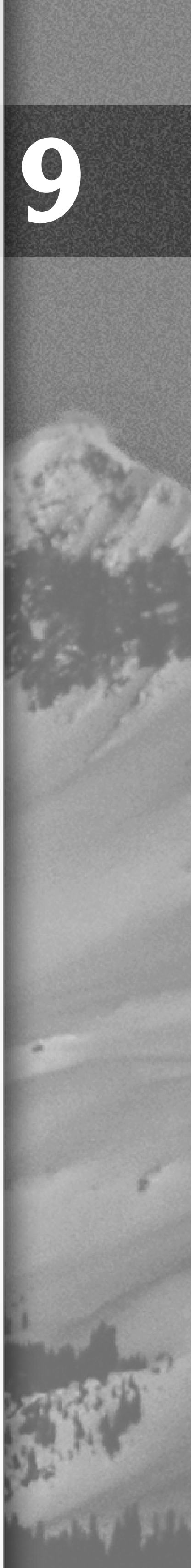
I've also covered other user-interface techniques, such as docking toolbars and other controls. You can consider this chapter the first step toward building professional applications. We will take other steps in the following chapters; but you already know enough to make your programs similar to some best-selling Windows applications, which may be very important for your clients.

Now that the elements of the main form of our programs are properly set up, we can consider adding secondary forms and dialog boxes. This is the topic of the next chapter, along with a general introduction to forms. The following chapter will then cover the overall structure of a Delphi application.

CHAPTER 9

Working with Forms

- Form styles, border styles, and border icons
- Mouse and keyboard input
- Painting and special effects
- Positioning, scaling, and scrolling forms
- Creating and closing forms
- Modal and modeless dialog boxes and forms
- Creating secondary forms dynamically
- Predefined dialog boxes
- Building a splash screen



If you've read the previous chapters, you should now be able to use Delphi's visual components to create the user interface of your applications. So let's turn our attention to another central element of development in Delphi: forms. We have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the TForm class are particularly interesting.

This chapter looks at some of the properties and styles of forms and at sizing and positioning them. I'll also introduce applications with multiple forms, the use of dialog boxes (custom and predefined ones), frames, and visual form inheritance. I'll also devote some time to input on a form, both from the keyboard and the mouse.

The *TForm* Class

Forms in Delphi are defined by the TForm class, included in the Forms unit of VCL. Of course, there is now a second definition of forms inside VisualCLX. Although I'll mainly refer to the VCL class in this chapter, I'll also try to highlight differences with the cross-platform version provided in CLX.

The TForm class is part of the windowed-controls hierarchy, which starts with the TWinControl (or TWidgetControl) class. Actually, TForm inherits from the *almost complete* TCustomForm, which in turn inherits from TScrollingWinControl (or TScrollingWidget). Having all of the features of their many base classes, forms have a long series of methods, properties, and events. For this reason, I won't try to list them here, but I'd rather present some interesting techniques related to forms throughout this chapter. I'll start by presenting a technique for *not* defining the form of a program at design time, using the TForm class directly, and then explore a few interesting properties of the form class.

Throughout the chapter, I'll point out a few differences between VCL forms and CLX forms. I've actually built a CLX version for most of the examples of this chapter, so you can immediately start experimenting with forms and dialog boxes in CLX, as well as VCL. As in past chapters, the CLX version of each example is prefixed by the letter *Q*.

Using Plain Forms

Generally, Delphi developers tend to create forms at design time, which implies deriving a new class from the base one, and build the content of the form visually. This is certainly a reasonable standard practice, but it is not compulsory to create a descendant of the TForm class to show a form, particularly if it is a simple one.

Consider this case: you have to show a rather long message (based on a string) to a user, and you don't want to use the simple predefined message box, as it will show up too large and not provide scroll bars. You can create a form with a memo component in it, and display the string inside it. Nothing prevents you from creating this form in the standard visual way, but you might consider doing this in code, particularly if you need a large degree of flexibility.

The DynaForm and QDynaForm examples (both on the companion CD), which are somewhat extreme, have no form defined at design time but include a unit with this function:

```
procedure ShowStringForm (str: string);
var
  form: TForm;
begin
  Application.CreateForm (TForm, form);
  form.caption := 'DynaForm';
  form.Position := poScreenCenter;
  with TMemo.Create (form) do
  begin
    Parent := form;
    Align := alClient;
    Scrollbars := ssVertical;
    ReadOnly := True;
    Color := form.Color;
    BorderStyle := bsNone;
    WordWrap := True;
    Text := str;
  end;
  form.Show;
end;
```

Besides the fact I had to create the form using the `Application` global object, a feature required by Delphi applications and discussed in the next chapter, this code simply does dynamically what you generally do with the form designer. Writing this code is undoubtedly more tedious, but it allows also a greater deal of flexibility, because any parameter can depend on external settings.

The `ShowStringForm` function above is not executed by an event of another form, as there are no traditional forms in this program. Instead, I've modified the project's source code to the following:

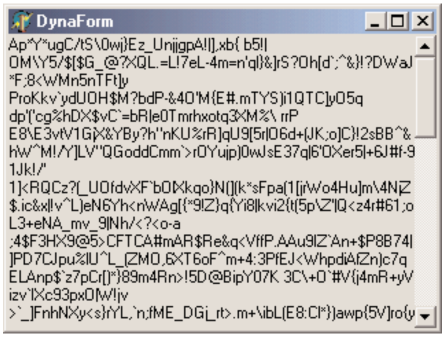
```
program DynaForm;

uses
  Forms,
  DynaMemo in 'DynaMemo.pas';

{$R *.RES}

var
```

```
  str: string;

begin
  str := '';
  Randomize;
  while Length (str) < 2000 do
    str := str + Char (32 + Random (94));
  ShowStringForm (str);

  Application.Run;
end.
```

The effect of running the DynaForm program is a strange-looking form filled with random characters (as you can see in Figure 9.1), not terribly useful in itself but for the idea it underscores.

FIGURE 9.1:

The dynamic form generated by the DynaForm example is completely created at run time, with no design-time support.



```
Ap*Y*ugC/tS\0wj}Ez_UnjjgpA!|],xb{ b5!|
0M\Y5/$[$G_@?XQL.=L!7eL-4m=n'q|}&]rS?0h[d`;^&}!?DWaJ
*F;8<WMn5nTFt]y
ProKkv`ydUOH$M?bdP-&40'M{E#.mTYS)i1QTC]y05q
dp'('cg%hDX$vC`=bR|e0Tmrhxotq3XM%\ rrP
E8\E3vtV1GjX&YBy?h"nKU%rR]qU9[5r|O6d+(JK;o]C}!2sBB^&
hW^M!/Y]LV"QGoddCmm`>r0Yujp)0wJsE37q|6'0Xer5|+6J#f-9
1Jk!/"
1]<RQCz?(_U0fdvXF`bOIXkqo}N()(k*sFpa(1[jrWo4Hu]m\4NjZ
$.ic&x|!v^L)eN6Yh<nWAg[{*9!Z}q{Yi8|kvi2{t(5p\Z'|Q<z4r#61;o
L3+eNA_mv_9|Nh/<?<o-a
;4$F3HX9@5>CFTCA#mAR$Re&q<VffP.AAu9|Z`An+$P8B74|
]PD7CJpu%|U^L_(ZM0,6XT6oF^m+4:3PfEJ<WhpdiAfZn)c7q
ELAnp$`z7pCr()*}89m4Rn>!5D@BipY07K 3C\+0`#V{j4mR+yV
izv`lXc93px0fW!jv
>`_]FnhNXy<s}rYL,`n;fME_DGj_rt>.m+\ibL(E8:Cl*})awp{5V]ro{y
```

TIP

An indirect advantage of this approach, compared to the use of DFM files for design-time forms, is that it would be much more difficult for an external programmer to grab information about the structure of the application. In Chapter 5 we saw that you can extract the DFM from the current Delphi executable file, but the same can be easily accomplished for any executable file compiled with Delphi for which you don't have the source code. If it is really important for you to keep to yourself a specific set of components you are using (maybe those in a specific form), and the default values of their properties, writing the extra code might be worth the effort.

The Form Style

The `FormStyle` property allows you to choose between a normal form (fsNormal) and the windows that make up a Multiple Document Interface (MDI) application. In this case, you'll use the fsMDIForm style for the MDI parent window—that is, the frame window of the MDI application—and the fsMDIChild style for the MDI child window. To know more about the development of an MDI application, look at Chapter 10.

A fourth option is the fsStayOnTop style, which determines whether the form has to always remain on top of all other windows, except for any that also happen to be "stay-on-top" windows.

To create a top-most form (a form whose window is always on top), you need only set the `FormStyle` property, as indicated above. This property has two different effects, depending on the kind of form you apply it to:

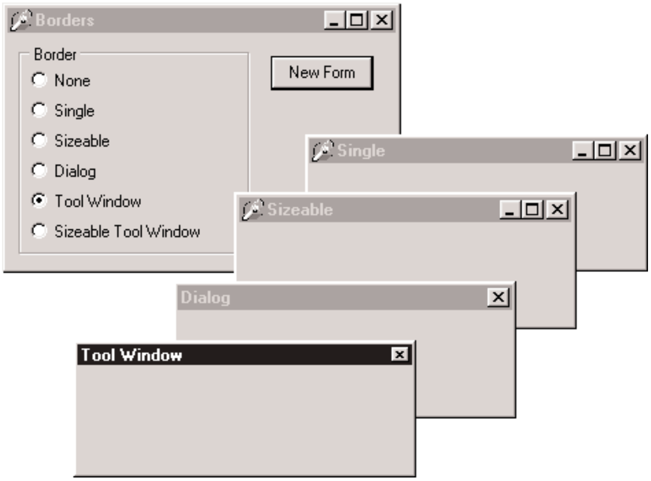
- The main form of an application will remain in front of every other application (unless other applications have the same top-most style, too). At times, this generates a rather ugly visual effect, so this makes sense only for special-purpose alert programs.
- A secondary form will remain in front of any other form of the application it belongs to. The windows of other applications are not affected, though. This is often used for floating toolbars and other forms that should stay in front of the main window.

The Border Style

Another important property of a form is its `BorderStyle`. This property refers to a visual element of the form, but it has a much more profound influence on the *behavior* of the window, as you can see in Figure 9.2.

FIGURE 9.2:

Sample forms with the various border styles, created by the Borders example



At design time, the form is always shown using the default value of the `BorderStyle` property, bsSizeable. This corresponds to a Windows style known as *thick frame*. When a main window has a thick frame around it, a user can resize it by dragging its border. This is made clear by the special *resize* cursors (with the shape of a double-pointer arrow) displayed when the user moves the mouse onto this thick window border.

A second important choice for this property is bsDialog. If you select it, the form uses as its border the typical dialog-box frame—a thick frame that doesn't allow resizing. In addition to this graphical element, note that if you select the bsDialog value, the form becomes a dialog box. This involves several changes. For example, the items on its system menu are different, and the form will ignore some of the elements of the `BorderIcons` set property.

WARNING Setting the `BorderStyle` property at design time produces no visible effect. In fact, several component properties do not take effect at design time, because they would prevent you from working on the component while developing the program. For example, how could you resize the form with the mouse if it were turned into a dialog box? When you run the application, though, the form will have the border you requested.

There are four more values we can assign to the `BorderStyle` property. The style bsSingle can be used to create a main window that's not resizable. Many games and applications based on windows with controls (such as data-entry forms) use this value, simply because resizing these forms makes no sense. Enlarging a form to see an empty area or reducing its size to make some components less visible often doesn't help a program's user (although Delphi's automatic scroll bars partially solve the last problem). The value bsNone is used only in very special situations and inside other forms. You'll never see an application with a main window that has no border or caption (except maybe as an example in a programming book to show you that it makes no sense).

The last two values, bsToolWindow and bsSizeToolWin, are related to the specific Win32 extended style `ws_ex_ToolWindow`. This style turns the window into a floating toolbox, with a small title font and close button. This style should not be used for the main window of an application.

To test the effect and behavior of the different values of the `BorderStyle` property, I've written a simple program called Borders, available also as QBorders in the CLX version. You've already seen its output, in Figure 9.2. However, I suggest you run this example and experiment with it for a while to understand all the differences in the forms.

WARNING In CLX, the enumeration for the `BorderStyle` property uses slightly different values, prefixed by the letters *fbs* (form border style). So we have fbsSingle, fbsDialog, and so on.

The main form of this program contains only a radio group and a button. There is also a secondary form, with no components and the `Position` property set to poDefaultPosOnly. This affects the initial position of the secondary form we'll create by clicking the button. (I'll discuss the `Position` property later in this chapter.)

The code of the program is very simple. When you click the button, a new form is dynamically created, depending on the selected item of the radio group:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);
var
  NewForm: TForm2;
begin
  NewForm := TForm2.Create (Application);
  NewForm.BorderStyle := TFormBorderStyle (BorderRadioGroup.ItemIndex);
  NewForm.Caption := BorderRadioGroup.Items[BorderRadioGroup.ItemIndex];
  NewForm.Show;
end;
```

This code actually uses a trick: it casts the number of the selected item into the TFormBorderStyle enumeration. This works because I've given the radio buttons the same order as the values of this enumeration:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsTolWindow,
    bsSizeToolWin);
```

The BtnNewFormClick method then copies the text of the radio button to the caption of the secondary form. This program refers to TForm2, the secondary form defined in a secondary unit of the program, saved as SECOND.PAS. For this reason, to compile the example, you must add the following lines to the implementation section of the unit of the main form:

```
uses
  Second;
```

TIP Whenever you need to refer to another unit of a program, place the corresponding uses statement in the implementation portion instead of the interface portion if possible. This speeds up the compilation process, results in cleaner code (because the units you include are separate from those included by Delphi), and prevents circular unit compilation errors. To accomplish this, you can also use the File ➢ Use Unit menu command.

The Border Icons

Another important element of a form is the presence of icons on its border. By default, a window has a small icon connected to the system menu, a Minimize button, a Maximize button, and a Close button on the far right. You can set different options using the BorderIcons property, a set with four possible values: biSystemMenu, biMinimize, biMaximize, and biHelp.

NOTE The biHelp border icon enables the "What's this?" Help. When this style is included and the biMinimize and biMaximize styles are excluded, a question mark appears in the form's title bar. If you click this question mark and then click a component inside the form (but not the form itself!), Delphi activates the Help about that object inside a pop-up window. This is demonstrated by the BIcons example, which has a simple Help file with a page connected to the `HelpContext` property of the button in the middle of the form.

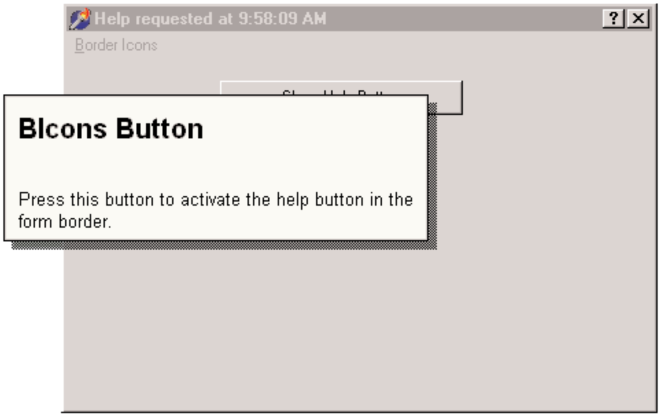
The BIcons example demonstrates the behavior of a form with different border icons and shows how to change this property at run time. The form of this example is very simple: It has only a menu, with a pull-down containing four menu items, one for each of the possible elements of the set of border icons. I've written a single method, connected with the four commands, that reads the check marks on the menu items to determine the value of the `BorderIcons` property. This code is therefore also a good exercise in working with sets:

```
procedure TForm1.SetIcons(Sender: TObject);
var
  BorIco: TBorderIcons;
begin
  (Sender as TMenuItem).Checked := not (Sender as TMenuItem).Checked;
  if SystemMenu1.Checked then
    BorIco := [biSystemMenu]
  else
    BorIco := [];
  if MaximizeBox1.Checked then
    Include (BorIco, biMaximize);
  if MinimizeBox1.Checked then
    Include (BorIco, biMinimize);
  if Help1.Checked then
    Include (BorIco, biHelp);
  BorderIcons := BorIco;
end;
```

While running the BIcons example, you can easily set and remove the various visual elements of the form's border. You'll immediately see that some of these elements are closely related: if you remove the system menu, all of the border icons will disappear; if you remove either the Minimize or Maximize button, it will be grayed; if you remove both these buttons, they will disappear. Notice also that in these last two cases, the corresponding items of the system menu are automatically disabled. This is the standard behavior for any Windows application. When the Maximize and Minimize buttons have been disabled, you can activate the Help button. As a shortcut to obtain this effect, you can click the button inside the form. Also, you can click the button after clicking the Help Menu icon to see a Help message, as you can see in Figure 9.3.

FIGURE 9.3:

The BIcons example. By selecting the help border icon and clicking over the button, you get the help displayed in the figure.



Help requested at 9:58:09 AM

BIcons Button

Press this button to activate the help button in the form border.

As an extra feature, the program also displays the time that the Help was invoked in the caption, by handling the `OnHelp` event of the form. This effect is visible in the figure.

WARNING By looking at the QBIcons version, built with CLX, you can clearly notice that a bug in the library prevents you from changing the border icons at run time, while the different design-time settings fully work.

Setting More Window Styles

The border style and border icons are indicated by two different Delphi properties, which can be used to set the initial value of the corresponding user interface elements. We have seen that besides changing the user interface, these properties affect the behavior of a window. It is important to know that in VCL (and obviously not in CLX), these border-related properties and the `FormStyle` property mainly correspond to different settings in the *style* and *extended style* of a window. These two terms reflect two parameters of the `CreateWindowEx` API function Delphi uses to create forms.

It is important to acknowledge this, because Delphi allows you to modify these two parameters freely by overriding the `CreateParams` virtual method:

```
public
  procedure CreateParams (var Params: TCreateParams); override;
```

This is the only way to use some of the peculiar window styles that are not directly available through form properties. For a list of window styles and extended styles, see the API Help under the topics “CreateWindow” and “CreateWindowEx.” You’ll notice that the Win32 API has styles for these functions, including those related to tool windows.

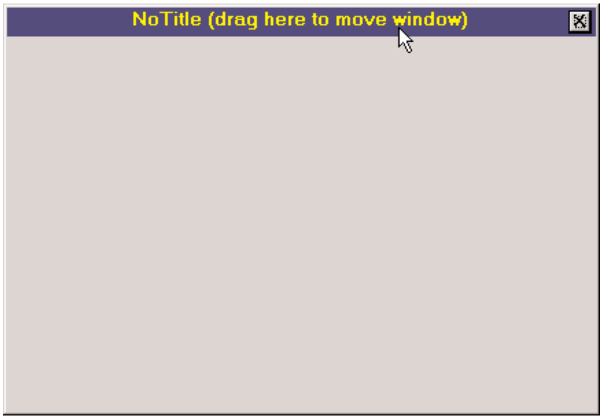
To show how to use this approach, I've written the NoTitle example on the companion CD, which lets you create a program with a custom caption. First we have to remove the standard caption but keep the resizing frame by setting the corresponding styles:

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := (Params.Style or ws_Popup) and not ws_Caption;
end;
```

NOTE Besides changing the style and other features of a window when it is created, you can change them at run time, although some of the settings do not take effect. To change most of the creation parameters at run time, you can use the `SetWindowLong` API function, which allows you to change the internal information of a window. The companion `GetWindowLong` function can be used to read the current status. Two more functions, `GetClassLong` and `SetClassLong`, can be used to read and modify class styles (the information of the `WindowClass` structure of `TCreateParams`). You'll seldom need to use these low-level Windows API functions in Delphi, unless you write advanced components.

To remove the caption, we need to change the overlapped style to a pop-up style; otherwise, the caption will simply stick. Now how do we add a custom caption? I've placed a label aligned to the upper border of the form and a small button on the far end. You can see this effect at run time in Figure 9.4.

FIGURE 9.4: The NoTitle example has no real caption but a fake one made with a label.



To make the fake caption work, we have to tell the system that a mouse operation on this area corresponds to a mouse operation on the caption. This can be done by intercepting the `wm_NCHitTest` Windows message, which is frequently sent to Windows to determine where the mouse currently is. When the hit is in the client area and on the label, we can pretend the mouse is on the caption by setting the proper result:

```
procedure TForm1.HitTest (var Msg: TWmNCHitTest);
  // message wm_NcHitTest
begin
  inherited;
  if (Msg.Result = htClient) and
    (Msg.YPos < Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then
    Msg.Result := htCaption;
end;
```

The GetSystemMetrics API function used in the listing above is used to query the operating system about the size of the various visual elements. It is important to make this request every time (and not cache the result) because users can customize most of these elements by using the Appearance page of the Desktop options (in Control Panel) and other Windows settings. The small button, instead, has a call to the Close method in its OnClick event handler. The button is kept in its position even when the window is resized by using the [akTop,akRight] value for the Anchors property. The form also has size constraints, so that a user cannot make it too small, as described in the "Form Constraints" section later in this chapter.

Direct Form Input

Having discussed some special capabilities of forms, I'll now move to a very important topic: user input in a form. If you decide to make limited use of components, you might write complex programs as well, receiving input from the mouse and the keyboard. In this chapter, I'll only introduce this topic.

Supervising Keyboard Input

Generally, forms don't handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. What you can do in these cases is turn on the KeyPreview property of the form. Then, even if you have some input controls, the form's OnKeyPress event will always be activated for any keyboard-input operation. The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character *0*, but the value 0 of the character set, indicated as #0).

The example I've built to demonstrate this, KPreview, has a form with no special properties (not even `KeyPreview`), a radio group with four options, and some edit boxes, as you can see in Figure 9.5.

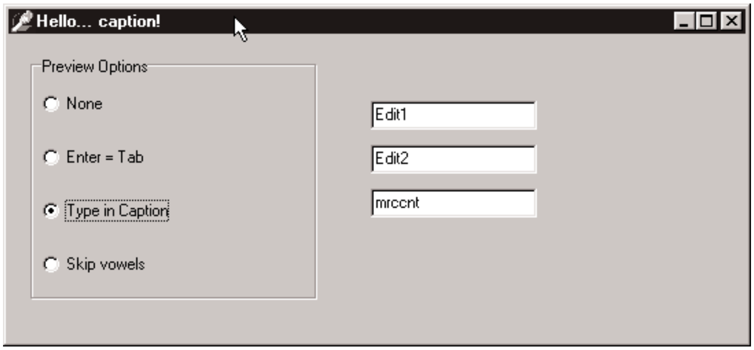


FIGURE 9.5:

The KPreview program allows you to type into the caption of the form (among other things).

By default the program does nothing special, except when the various radio buttons are used to enable the key preview:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
  KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```

Now we'll start receiving the `OnKeyPress` events, and we can do one of the three actions requested by the three special buttons of the radio group. The action depends on the value of the `ItemIndex` property of the radio group component. This is the reason the event handler is based on a `case` statement:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  case RadioPreview.ItemIndex of
    ...
```

In the first case, if the value of the `Key` parameter is `#13`, which corresponds to the Enter key, we disable the operation (setting `Key` to zero) and then mimic the activation of the Tab key. There are many ways to accomplish this, but the one I've chosen is quite particular. I send the `CM_DialogKey` message to the form, passing the code for the Tab key (`VK_TAB`):

```
    1: // Enter = Tab
      if Key = #13 then
      begin
        Key := #0;
        Perform (CM_DialogKey, VK_TAB, 0);
      end;
```

NOTE The CM_DialogKey message is an internal, undocumented Delphi message. There are a few of them, actually quite interesting to build advanced components for and for some special coding, but Borland never described those. For more information on this topic, refer to the sidebar "Component Messages and Notifications" in Chapter 11. Notice also that this exact message-based coding style is not available under CLX.

To type in the caption of the form, the program simply adds the character to the current Caption. There are two special cases. When the Backspace key is pressed, the last character of the string is removed (by copying to the Caption all the characters of the current Caption but the last one). When the Enter key is pressed, the program stops the operation, by resetting the ItemIndex property of the radio group control. Here is the code:

```
2: // type in caption
begin
  if Key = #8 then // backspace: remove last char
    Caption := Copy (Caption, 1, Length (Caption) - 1)
  else if Key = #13 then // enter: stop operation
    RadioPreview.ItemIndex := 0
  else // anything else: add character
    Caption := Caption + Key;
  Key := #0;
end;
```

Finally, if the last radio item is selected, the code checks whether the character is a vowel (by testing for its inclusion in a constant "vowel set"). In this case, the character is skipped altogether:

```
3: // skip vowels
  if Key in ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'] then
    Key := #0;
```

Getting Mouse Input

When a user clicks one of the mouse buttons over a form (or over a component, by the way), Windows sends the application some messages. Delphi defines some events you can use to write code that responds to these messages. The two basic events are OnMouseDown, received when a mouse button is clicked, and OnMouseUp, received when the button is released. Another fundamental system message is related to mouse movement; the event is OnMouseMove. Although it should be easy to understand the meaning of the three messages—down, up, and move—the question that might arise is, how do they relate to the OnClick event we have often used up to now?

We have used the OnClick event for components, but it is also available for the form. Its general meaning is that the left mouse button has been clicked and released on the same window or component. However, between these two actions, the cursor might have been moved outside the area of the window or component, while the left mouse button was held down.

Another difference between the `OnMouseXX` and `OnClick` events is that the latter relates only to the *left* mouse button. Most of the mouse types connected to a Windows PC have two mouse buttons, and some even have three. Usually we refer to these buttons as the left mouse button, generally used for selection; the right mouse button, for local menus; and the middle mouse button, seldom used. Nowadays most new mouse devices have a "button wheel" instead of the middle button. Users typically use the wheel for scrolling (causing an `OnMouseWheel` event), but they can also press it (generating the `OnMouseWheelDown` and `OnMouseWheelUp` events). Mouse wheel events are automatically converted into scrolling events.

Using Windows without a Mouse

A user should always be able to use any Windows application without the mouse. This is not an option; it is a Windows programming rule. Of course, an application might be easier to use with a mouse, but that should never be mandatory. In fact, there are users who for various reasons might not have a mouse connected, such as travelers with a small laptop and no space, workers in industrial environments, and bank clerks with other peripherals around.

There is another reason to support the keyboard: Using the mouse is nice, but it tends to be slower. If you are a skilled touch typist, you won't use the mouse to drag a word of text; you'll use shortcut keys to copy and paste it, without moving your hands from the keyboard.

For all these reasons, you should always set up a proper tab order for a form's components, remember to add keys for buttons and menu items for keyboard selection, use shortcut keys on menu commands, and so on.

The Parameters of the Mouse Events

All of the lower-level mouse events have the same parameters: the usual `Sender` parameter; a `Button` parameter indicating which of the three mouse buttons has been clicked (mbRight, mbLeft, or mbCenter); the `Shift` parameter indicating which of the *mouse-related keys* (Alt, Ctrl, and Shift, plus the three mouse buttons themselves) were pressed when the event occurred; and the `x` and `y` coordinates of the position of the mouse, in *client area* coordinates of the current window.

Using this information, it is very simple to draw a small circle in the position of a left mouse button–down event:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

NOTE To draw on the form, we use a very special property: `Canvas`. A `TCanvas` object has two distinctive features: it holds a collection of drawing tools (such as a pen, a brush, and a font) and it has some drawing methods, which use the current tools. The kind of direct drawing code in this example is not correct, because the on-screen image is not persistent: moving another window over the current one will clear its output. The next example demonstrates the Windows "store-and-draw" approach.

Dragging and Drawing with the Mouse

To demonstrate a few of the mouse techniques discussed so far, I've built a simple example based on a form without any component and called MouseOne in the VCL version and QMouseOne in the CLX version. The first feature of this program is that it displays in the `Caption` of the form the current position of the mouse:

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
end;
```

You can use this simple feature of the program to better understand how the mouse works. Make this test: run the program (this simple version or the complete one) and resize the windows on the desktop so that the form of the MouseOne or QMouseOne program is behind another window and inactive but with the title visible. Now move the mouse over the form, and you'll see that the coordinates change. This means that the `OnMouseMove` event is sent to the application even if its window is not active, and it proves what I have already mentioned: Mouse messages are always directed to the window under the mouse. The only exception is the mouse capture operation I'll discuss in this same example.

Besides showing the position in the title of the window, the MouseOne/QMouseOne example can track mouse movements by painting small pixels on the form if the user keeps the Shift key pressed. (Again this direct painting code produces non-persistent output.)

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
```

```
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
  end;
```

TIP The `TCanvas` class of the CLX library doesn't include a `Pixels` array. Instead, you can call the `DrawPoint` method after setting a proper color for the pen, as I've done in the QMouseOne example.

The real feature of this example, however, is the direct mouse-dragging support. Contrary to what you might think, Windows has no system support for dragging, which is implemented in VCL by means of lower-level mouse events and operations. (An example of dragging from one control to another was discussed in the last chapter.) In VCL, forms cannot originate dragging operations, so in this case we are obliged to use the low-level approach. The aim of this example is to draw a rectangle from the initial position of the dragging operation to the final one, giving the users some visual clue of the operation they are doing.

The idea behind dragging is quite simple. The program receives a sequence of button-down, mouse-move, and button-up messages. When the button is clicked, dragging begins, although the real actions take place only when the user moves the mouse (without releasing the mouse button) and when dragging terminates (when the button-up message arrives). The problem with this basic approach is that it is not reliable. A window usually receives mouse events only when the mouse is over its client area; so if the user clicks the mouse button, moves the mouse onto another window, and then releases the button, the second window will receive the button-up message.

There are two solutions to this problem. One (seldom used) is mouse clipping. Using a Windows API function (namely `ClipCursor`), you can force the mouse not to leave a certain area of the screen. When you try to move it outside the specified area, it stumbles against an invisible barrier. The second and more common solution is to capture the mouse. When a window captures the mouse, all the subsequent mouse input is sent to that window. This is the approach we will use for the MouseOne/QMouseOne example.

The code of the example is built around three methods: `FormMouseDown`, `FormMouseMove`, and `FormMouseUp`. Clicking the left mouse button over the form starts the process, setting the `fDragging` Boolean field of the form (which indicates that dragging is in action in the other two methods). The method also uses a `TRect` variable used to keep track of the initial and current position of the dragging. Here is the code:

```
procedure TMouseForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    fDragging := True;
    Mouse.Capture := Handle;
    fRect.Left := X;
    fRect.Top := Y;
    fRect.BottomRight := fRect.TopLeft;
    Canvas.DrawFocusRect (fRect);
  end;
end;
```

An important action of this method is the call to the SetCapture API function, obtained by setting the Capture property of the global object Mouse. Now even if a user moves the mouse outside of the client area, the form still receives all mouse-related messages. You can see that for yourself by moving the mouse toward the upper-left corner of the screen; the program shows negative coordinates in the caption.

TIP The global Mouse object allows you to get global information about the mouse, such as its presence, its type, and the current position, as well as set some of its global features. This global object hides a few API functions, making your code simpler and more portable.

When dragging is active and the user moves the mouse, the program draws a dotted rectangle corresponding to the actual position. Actually, the program calls the DrawFocusRect method twice. The first time this method is called, it deletes the current image, thanks to the fact that two consecutive calls to DrawFocusRect simply reset the original situation. After updating the position of the rectangle, the program calls the method a second time:

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
    Canvas.DrawFocusRect (fRect);
    fRect.Right := X;
    fRect.Bottom := Y;
    Canvas.DrawFocusRect (fRect);
  end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
end;
```

When the mouse button is released, the program terminates the dragging operation by resetting the Capture property of the Mouse object, which internally calls the ReleaseCapture API function, and by setting the value of the fDragging field to False:

```
procedure TMouseForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
  begin
    Mouse.Capture := 0; // calls ReleaseCapture
    fDragging := False;
    Invalidate;
  end;
end;
```

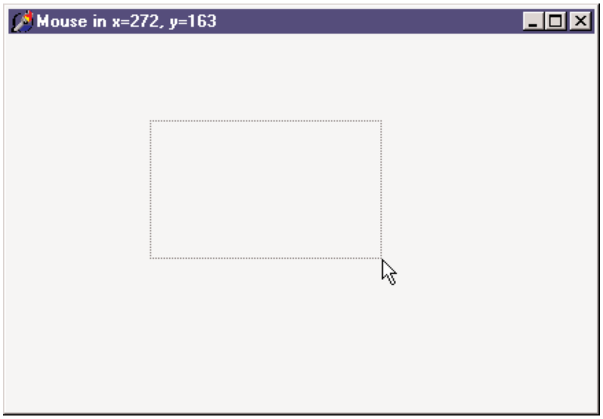
The final call, Invalidate, triggers a painting operation and executes the following OnPaint event handler:

```
procedure TMouseForm.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle (fRect.Left, fRect.Top, fRect.Right, fRect.Bottom);
end;
```

This makes the output of the form persistent, even if you hide it behind another form. Figure 9.6 shows a previous version of the rectangle and a dragging operation in action.

FIGURE 9.6:

The MouseOne example uses a dotted line to indicate, during a dragging operation, the final area of a rectangle.



TIP

Under Qt, there are no Windows handles, but the Capture property of the mouse is still available. You assign to it, however, the object of the component that has to capture the mouse (for example, Self to indicate the form), or set the property to nil to release it. You can see this code in the QMouseOne example.

Painting in Windows

Why do we need to handle the OnPaint event to produce a proper output, and why can't we paint directly over the form canvas? It depends on Windows' default behavior. As you draw on a window, Windows does *not* store the resulting image. When the window is covered, its contents are usually lost.

The reason for this behavior is simple: to save memory. Windows assumes it's "cheaper" in the long run to redraw the screen using code than to dedicate system memory to preserving the display state of a window. It's a classic memory-versus-CPU-cycles trade-off. A color bitmap for a 300×400 image at 256 colors requires about 120 KB. By increasing the color count or the number of pixels, you can easily have full-screen bitmaps of about 1 MB and reach 4 MB of memory for a 1280×1024 resolution at 16 million colors. If storing the bitmap was the default choice, running half a dozen simple applications would require at least 8 MB of memory, if not 16 MB, just for remembering their current output.

In the event that you want to have a consistent output for your applications, there are two techniques you can use. The general solution is to store enough data about the output to be able to reproduce it when the system sends a *painting* requested. An alternative approach is to save the output of the form in a bitmap while you produce it, by placing an Image component over the form and drawing on the canvas of this image component.

The first technique, painting, is the common approach to handling output in Windows, aside from particular graphics-oriented programs that store the form's whole image in a bitmap. The approach used to implement painting has a very descriptive name: *store and paint.* In fact, when the user clicks a mouse button or performs any other operation, we need to store the position and other elements; then, in the painting method, we use this information to actually paint the corresponding image.

The idea of this approach is to let the application repaint its whole surface under any of the possible conditions. If we provide a method to redraw the contents of the form, and if this method is automatically called when a portion of the form has been hidden and needs repainting, we will be able to re-create the output properly.

Since this approach takes two steps, we must be able to execute these two operations in a row, asking the system to repaint the window—without waiting for the system to ask for this. You can use several methods to invoke repainting: Invalidate, Update, Repaint, and Refresh. The first two correspond to the Windows API functions, while the latter two have been introduced by Delphi.

- The Invalidate method informs Windows that the entire surface of the form should be repainted. The most important thing is that Invalidate does *not* enforce a painting operation immediately. Windows simply stores the request and will respond to it only after the current procedure has been completely executed and as soon as there are no other events pending in the system. Windows deliberately delays the painting operation because it is one of the most time-consuming operations. At times, with this delay, it is possible to paint the form only after multiple changes have taken place, avoiding multiple consecutive calls to the (slow) paint method.

- The `Update` method asks Windows to update the contents of the form, repainting it immediately. However, remember that this operation will take place only if there is an *invalid area*. This happens if the `Invalidate` method has just been called or as the result of an operation by the user. If there is no invalid area, a call to `Update` has no effect at all. For this reason, it is common to see a call to `Update` just after a call to `Invalidate`. This is exactly what is done by the two Delphi methods, `Repaint` and `Refresh`.
- The `Repaint` method calls `Invalidate` and `Update` in sequence. As a result, it activates the `OnPaint` event immediately. There is a slightly different version of this method called `Refresh`. For a form the effect is the same; for components it might be slightly different.

When you need to ask the form for a repaint operation, you should generally call `Invalidate`, following the standard Windows approach. This is particularly important when you need to request this operation frequently, because if Windows takes too much time to update the screen, the requests for repainting can be accumulated into a simple repaint action. The `wm_Paint` message in Windows is a sort of low-priority message. To be more precise, if a request for repainting is pending but other messages are waiting, the other messages are handled before the system actually performs the paint action.

On the other hand, if you call `Repaint` several times, the screen must be repainted each time before Windows can process other messages, and because paint operations are computationally intensive, this can actually make your application less responsive. There are times, however, when you want the application to repaint a surface as quickly as possible. In these less-frequent cases, calling `Repaint` is the way to go.

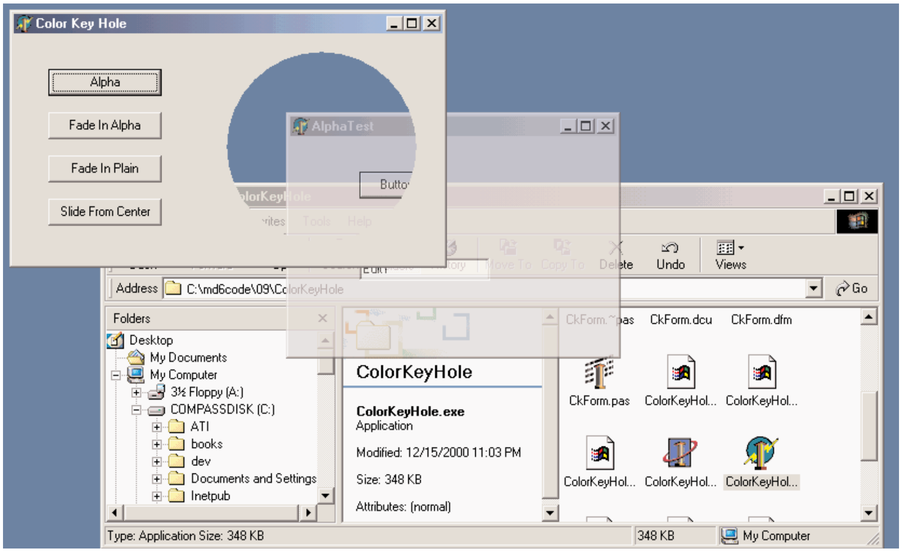
NOTE Another important consideration is that during a paint operation Windows redraws only the so-called *update region*, to speed up the operation. For this reason if you invalidate only a portion of a window, only that area will be repainted. To accomplish this you can use the `InvalidateRect` and `InvalidateRegion` functions. Actually, this feature is a double-edged sword. It is a very powerful technique, which can improve speed and reduce the flickering caused by frequent repaint operations. On the other hand, it can also produce incorrect output. A typical problem is when only some of the areas affected by the user operations are actually modified, while others remain as they were even if the system executes the source code that is supposed to update them. In fact, if a painting operation falls outside the update region, the system ignores it, as if it were outside the visible area of a window.

Unusual Techniques: Alpha Blending, Color Key, and the Animate API

One of the few new features of Delphi 6 related to forms is support for some new Windows APIs regarding the way forms are displayed (not available under Qt/CLX). For a form, *alpha blending* allows you to merge the content of a form with what's behind it on the screen, something you'll rarely need, at least in a business application. The technique is certainly more interesting when applied to bitmap (with the new `AlphaBlend` and `AlphaDIBBlend` API functions) than to a form itself. In any case, by setting the `AlphaBlend` property of a form to True and giving to the `AlphaBlendValue` property a value lower than 255, you'll be able to see, in transparency, what's behind the form. The lower the `AlphaBlendValue`, the more the form will *fade*. You can see an example of alpha blending in Figure 9.7, taken from the CkKeyHole example

FIGURE 9.7:

The output of the CkKeyHole, showing the effect of the new `TransparentColor` and `AlphaBlend` properties, and also the AnimateWindow API.



Address C:\md6code\09\ColorKeyHole

ColorKeyHole

ColorKeyHole.exe
Application

Modified: 12/15/2000 11:03 PM

Size: 348 KB

Attributes: (normal)

Type: Application Size: 348 KB 348 KB

This is not the only new Delphi feature in the area of what I can only call *unusual*. The second is the new `TransparentColor` property, which allows you to indicate a transparent color, which will be replaced by the background, creating a sort of hole in a form. The transparent color is indicated by the `TransparentColorValue` property. Again, you can see an example of this effect in Figure 9.7.

Finally, you can use a native Windows technique, animated display, which is not directly supported by Delphi (beyond the display of hints). For example, instead of calling the Show method of a form, you can write:

```
Form3.Hide;
AnimateWindow (Form3.Handle, 2000, AW_BLEND);
Form3.Show;
```

Notice you have to call the Show method at the end for the form to behave properly. A similar animation effect can also be obtained by changing the AlphaBlendValue in a loop. The AnimateWindow API can also be used to obtain the display of the form starting from the center (with the AW_CENTER flag) or from one of its sides (AW_HOR_POSITIVE, AW_HOR_NEGATIVE, AW_VER_POSITIVE, or AW_VER_NEGATIVE), as is common for slide shows.

This same function can also be applied to windowed controls, obtaining a fade-in effect instead of the usual direct appearance. I keep having serious doubts about the waste of CPU cycles these animations cause, but I have to say that if they are applied properly and in the right program, they can improve the user interface.

Position, Size, Scrolling, and Scaling

Once you have designed a form in Delphi, you run the program, and you expect the form to show up exactly as you prepared it. However, a user of your application might have a different screen resolution or might want to resize the form (if this is possible, depending on the border style), eventually affecting the user interface. We've already discussed (mainly in Chapter 7) some techniques related to controls, such as alignment and anchors. Here I want to specifically address elements related to the form as a whole.

Besides differences in the user system, there are many reasons to change Delphi defaults in this area. For example, you might want to run two copies of the program and avoid having all the forms show up in exactly the same place. I've collected many other related elements, including form scrolling, in this portion of the chapter.

The Form Position

There are a few properties you can use to set the position of a form. The Position property indicates how Delphi determines the initial position of the form. The default poDesigned value indicates that the form will appear where you designed it and where you use the positional (Left and Top) and size (Width and Height) properties of the form.

Some of the other choices (poDefault, poDefaultPosOnly, and poDefaultSizeOnly) depend on a feature of the operating system: using a specific flag, Windows can position and/or size new windows using a cascade layout. In this way, the positional and size properties you set at design time will be ignored, but running the application twice you won't get overlapping windows. The default positions are ignored when the form has a dialog border style.

Finally, with the poScreenCenter value, the form is displayed in the center of the screen, with the size you set at design time. This is a very common setting for dialog boxes and other secondary forms.

Another property that affects the initial size and position of a window is its *state*. You can use the WindowState property at design time to display a maximized or minimized window at startup. This property, in fact, can have only three values: wsNormal, wsMinimized, and wsMaximized. The meaning of this property is intuitive. If you set a minimized window state, at startup the form will be displayed in the Windows Taskbar. For the main form of an application, this property can be automatically set by specifying the corresponding attributes in a shortcut referring to the application.

Of course, you can maximize or minimize a window at run time, too. Simply changing the value of the WindowState property to wsMaximized or to wsNormal produces the expected effect. Setting the property to wsMinimized, however, creates a minimized window that is placed over the Taskbar, not within it. This is not the expected action for a main form, but for a secondary form! The simple solution to this problem is to call the Minimize method of the Application object. There is also a Restore method in the TApplication class that you can use when you need to restore a form, although most often the user will do this operation using the Restore command of the system menu.

The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the Width and Height properties or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the OnResize event, where you can perform custom actions to adapt the user interface to the new size of the form).

However, if you look at a form's properties in source code or in the online Help, you can see that there are two properties referring to its width and two referring to its height. Height and Width refer to the size of the form, including the borders; ClientHeight and ClientWidth refer to the size of the internal area of the form, excluding the borders, caption, scroll bars (if any), and menu bar. The client area of the form is the surface you can use to place components on the form, to create output, and to receive user input.

Since you might be interested in having a certain available area for your components, it often makes more sense to set the client size of a form instead of its global size. This is straightforward, because as you set one of the two client properties, the corresponding form property changes accordingly.

TIP

In Windows, it is also possible to create output and receive input from the nonclient area of the form—that is, its border. Painting on the border and getting input when you click it are complex issues. If you are interested, look in the Help file at the description of such Windows messages as `wm_NCPaint`, `wm_NCCalcSize`, and `wm_NCHitTest` and the series of nonclient messages related to the mouse input, such as `wm_NCLButtonDown`. The difficulty of this approach is in combining your code with the default Windows behavior.

Form Constraints

When you choose a resizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the `wm_Size` message, which generates the `OnResize` event. `OnResize` takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

Delphi provides a specific property for forms and also for all controls: the `Constraints` property. Simply setting the subproperties of the `Constraints` property to the proper maximum and minimum values creates a form that cannot be resized beyond those limits. Here is an example:

```
object Form1: TForm1
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

Notice that as you set up the `Constraints` property, it has an immediate effect even at design time, changing the size of the form if it is outside the permitted area.

Delphi also uses the maximum constraints for maximized windows, producing an awkward effect. For this reason, you should generally disable the Maximize button of a window that has a maximum size. There are cases in which maximized windows with a limited size make sense—this is the behavior of Delphi's main window. In case you need to change constraints at run time, you can also consider using two specific events, `OnCanResize` and `OnConstrainedResize`. The first of the two can also be used to disable resizing a form or control in given circumstances.

Scrolling a Form

When you build a simple application, a single form might hold all of the components you need. As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms. If you reduce the space occupied by the components, you might add some capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen (or at least bigger than its visible portion on the screen).

Adding a scroll bar to a form is simple. In fact, you don't need to do anything. If you place several components in a big form and then reduce its size, a scroll bar will be added to the form automatically, as long as you haven't changed the value of the AutoScroll property from its default of True.

Along with AutoScroll, forms have two properties, HorzScrollBar and VertScrollBar, which can be used to set several properties of the two TFormScrollBar objects associated with the form. The Visible property indicates whether the scroll bar is present, the Position property determines the initial status of the scroll thumb, and the Increment property determines the effect of clicking one of the arrows at the ends of the scroll bar. The most important property, however, is Range.

The Range property of a scroll bar determines the virtual size of the form, not the actual range of values of the scroll bar. Suppose you need a form that will host several components and will therefore need to be 1000 pixels wide. We can use this value to set the "virtual range" of the form, changing the Range of the horizontal scroll bar.

The Position property of the scroll bar will range from 0 to 1000 minus the current size of the client area. For example, if the client area of the form is 300 pixels wide, you can scroll 700 pixels to see the far end of the form (the thousandth pixel).

A Scroll Testing Example

To demonstrate the specific case I've just discussed, I've built the Scroll1 example, which has a virtual form 1000 pixels wide. To accomplish this, I've set the range of the horizontal scroll bar to 1000:

```
object Form1: TForm1
  Width = 458
  Height = 368
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  Caption = 'Scrolling Form'
  OnResize = FormResize
  ...
```

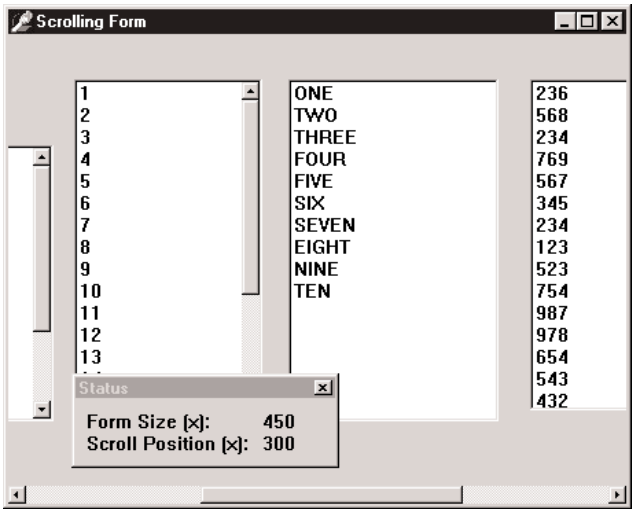
The form of this example has been filled with meaningless list boxes, and I could have obtained the same scroll-bar range by placing the right-most list box so that its position (Left) plus its size (Width) would equal 1000.

The interesting part of the example is the presence of a toolbox window displaying the status of the form and of its horizontal scroll bar. This second form has four labels; two with fixed text and two with the actual output. Besides this, the secondary form (called Status) has a bsToolWindow border style and is a top-most window. You should also set its Visible property to True, to have its window automatically displayed at startup:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  Caption = 'Status'
  FormStyle = fsStayOnTop
  Visible = True
  object Label1: TLabel...
  ...
```

There isn't much code in this program. Its aim is to update the values in the toolbox each time the form is resized or scrolled (as you can see in Figure 9.8). The first part is extremely simple. You can handle the OnResize event of the form and simply copy a couple of values to the two labels. The labels are part of another form, so you need to prefix them with the name of the form instance, Status:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```



```
1     ONE     236
2     TWO     568
3     THREE   234
4     FOUR    769
5     FIVE    567
6     SIX     345
7     SEVEN   234
8     EIGHT   123
9     NINE    523
10    TEN     754
11            987
12            978
13            654
              543
              432
```

Form Size (x): 450
Scroll Position (x): 300

FIGURE 9.8:

The output of the Scroll1 example

If we wanted to change the output each time the user scrolls the contents of the form, we could not use a Delphi event handler, because there isn't an OnScroll event for forms (although there is one for stand-alone ScrollBar components). Omitting this event makes sense, because Delphi forms handle scroll bars automatically in a powerful way. In Windows, by contrast, scroll bars are extremely low-level elements, requiring a lot of coding. Handling the scroll event makes sense only in special cases, such as when you want to keep track precisely of the scrolling operations made by a user.

Here is the code we need to write. First, add a method declaration to the class and associate it with the Windows horizontal scroll message (wm_HScroll):

```
public
  procedure FormScroll (var ScrollData: TWMScroll);
    message wm_HScroll;
```

Then write the code of this procedure, which is almost the same as the code of the FormResize method we've seen before:

```
procedure TForm1.FormScroll (var ScrollData: TWMScroll);
begin
  inherited;
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

It's important to add the call to inherited, which activates the method related to the same message in the base class form. The inherited keyword in Windows message handlers calls the method of the base class we are overriding, which is the one associated with the corresponding Windows message (even if the procedure name is different). Without this call, the form won't have its default scrolling behavior; that is, it won't scroll at all.

NOTE Because in CLX you cannot handle the low-level scroll messages, there seems to be no easy way to create a program similar to Scroll1. This isn't terribly important in real-world applications, as the scrolling system is automatic, and can probably be accomplished by hooking in the CLX library at a lower level.

Automatic Scrolling

The scroll bar's Range property can seem strange until you start to use it consistently. When you think about it a little, you'll start to understand the advantages of the "virtual range" approach. First of all, the scroll bar is automatically removed from the form when the client area of the form is big enough to accommodate the virtual size; and when you reduce the size of the form, the scroll bar is added again.

This feature becomes particularly interesting when the AutoScroll property of the form is set to True. In this case, the extreme positions of the right-most and lower controls are automatically copied into the Range properties of the form's two scroll bars. Automatic scrolling works well in Delphi. In the last example, the virtual size of the form would be set to the right border of the last list box. This was defined with the following attributes:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Therefore, the horizontal virtual size of the form would be 977 (the sum of the two preceding values). This number is automatically copied into the Range field of the HorzScrollBar property of the form, unless you change it manually to have a bigger form (as I've done for the Scroll1 example, setting it to 1000 to leave some space between the last list box and the border of the form). You can see this value in the Object Inspector, or make the following test: run the program, size the form as you like, and move the scroll thumb to the right-most position. When you add the size of the form and the position of the thumb, you'll always get 1000, the virtual coordinate of the right-most pixel of the form, whatever the size.

Scrolling and Form Coordinates

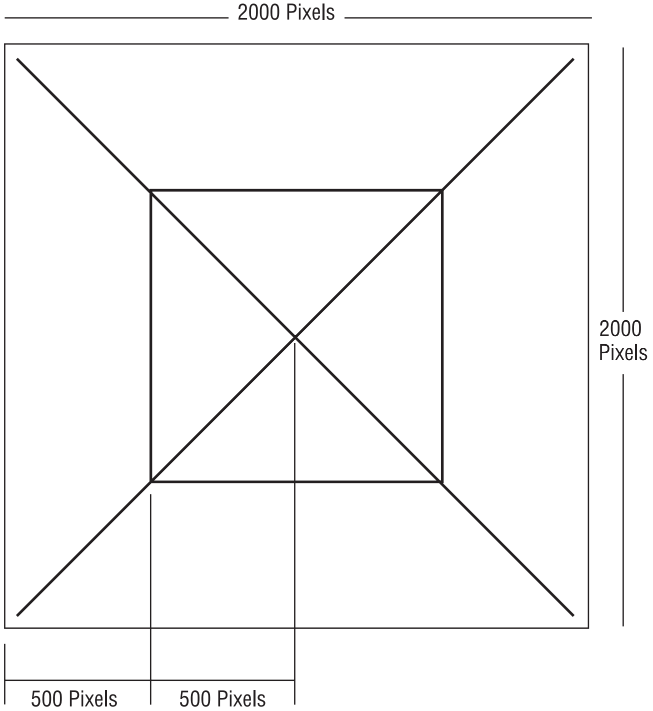
We have just seen that forms can automatically scroll their components. But what happens if you paint directly on the surface of the form? Some problems arise, but their solution is at hand. Suppose that we want to draw some lines on the virtual surface of a form, as shown in Figure 9.9.

Since you probably do not own a monitor capable of displaying 2000 pixels on each axis, you can create a smaller form, add two scroll bars, and set their Range property, as I've done in the Scroll2 example. Here is the textual description of the form:

```
object Form1: TForm1
  HorzScrollBar.Range = 2000
  VertScrollBar.Range = 2000
  ClientHeight = 336
  ClientWidth = 472
  OnPaint = FormPaint
end
```

FIGURE 9.9:

The lines to draw on the virtual surface of the form



If we simply draw the lines using the virtual coordinates of the form, the image won't display properly. In fact, in the `OnPaint` response method, we need to compute the virtual coordinates ourselves. Fortunately, this is easy, since we know that the virtual `X1` and `Y1` coordinates of the upper-left corner of the client area correspond to the current positions of the two scroll bars:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  X1, Y1: Integer;
begin
  X1 := HorzScrollBar.Position;
  Y1 := VertScrollBar.Position;

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30-X1, 30-Y1);
  Canvas.LineTo (1970-X1, 1970-Y1);
// and so on ...
```

As a better alternative, instead of computing the proper coordinate for each output operation, we can call the SetWindowOrgEx API to move the origin of the coordinates of the Canvas itself. This way, our drawing code will directly refer to virtual coordinates but will be displayed properly:

```
procedure TForm2.FormPaint(Sender: TObject);
begin
  SetWindowOrgEx (Canvas.Handle, HorzScrollbar.Position,
    VertScrollbar.Position, nil);

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30, 30);
  Canvas.LineTo (1970, 1970);

  // and so on ...
```

This is the version of the program you'll find in the source code on the CD. Try using the program and commenting out the SetWindowOrgEx call to see what happens if you don't use virtual coordinates: You'll find that the output of the program is not correct—it won't scroll, and the same image will always remain in the same position, regardless of scrolling operations. Notice also that the Qt/CLX version of the program, called QScroll2, doesn't use virtual coordinates but simply subtracts the scroll positions from each of the hard-coded coordinates.

Scaling Forms

When you create a form with multiple components, you can select a fixed size border or let the user resize the form and automatically add scroll bars to reach the components falling outside the visible portion of the form, as we've just seen. This might also happen because a user of your application has a display driver with a much smaller number of pixels than yours.

Instead of simply reducing the form size and scrolling the content, you might want to reduce the size of each of the components at the same time. This automatically happens also if the user has a system font with a different pixel-per-inch ratio than the one you used for development. To address these problems, Delphi has some nice scaling features, but they aren't fully intuitive.

The form's ScaleBy method allows you to scale the form and each of its components. The PixelsPerInch and Scaled properties allow Delphi to resize an application automatically when the application is run with a different system font size, often because of a different screen resolution. In both cases, to make the form scale its window, be sure to also set the

AutoScroll property to False. Otherwise, the contents of the form will be scaled, but the form border itself will not. These two approaches are discussed in the next two sections.

NOTE Form scaling is calculated based on the difference between the font height at run time and the font height at design time. Scaling ensures that edit and other controls are large enough to display their text using the user's font preferences without clipping the text. The form scales as well, as we will see later on, but the main point is to make edit and other controls readable.

Manual Form Scaling

Any time you want to scale a form, including its components, you can use the ScaleBy method, which has two integer parameters, a multiplier and a divisor—it's a fraction. For example, with this statement the size of the current form is reduced to three-quarters of its original size:

```
ScaleBy (3, 4);
```

Generally, it is easier to use percentage values. The same effect can be obtained by using:

```
ScaleBy (75, 100);
```

When you scale a form, all the proportions are maintained, but if you go below or above certain limits, the text strings can alter their proportions slightly. The problem is that in Windows, components can be placed and sized only in whole pixels, while scaling almost always involves multiplying by fractional numbers. So any fractional portion of a component's origin or size will be truncated.

I've built a simple example, Scale or QScale, to show how you can scale a form manually, responding to a request by the user. The form of this application (see Figure 9.10) has two buttons, a label, an edit box, and an UpDown control connected to it (via its Associate property). With this setting, a user can type numbers in the edit box or click the two small arrows to increase or decrease the value (by the amount indicated by the Increment property). To extract the input value, you can use the Text property of the edit box or the Position of the UpDown control.

When you click the Do Scale button, the current input value is used to determine the scaling percentage of the form:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);
begin
  AmountScaled := UpDown1.Position;
  ScaleBy (AmountScaled, 100);
  UpDown1.Height := Edit1.Height;
  ScaleButton.Enabled := False;
  RestoreButton.Enabled := True;
end;
```

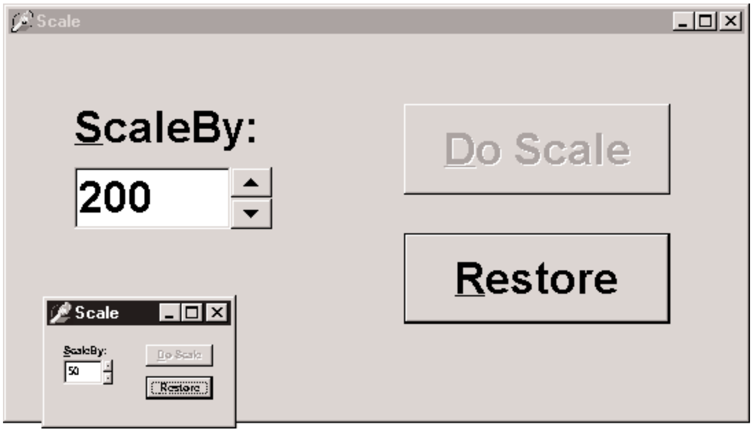


FIGURE 9.10:
The form of the Scale example after a scaling with 50 and 200

This method stores the current input value in the form's `AmountScaled` private field and enables the Restore button, disabling the one that was clicked. Later, when the user clicks the Restore button, the opposite scaling takes place. By having to restore the form before another scaling operation takes place, I avoid an accumulation of round-off errors. I've added also a line to set the `Height` of the UpDown component to the same `Height` as the edit box it is attached to. This prevents small differences between the two, due to scaling problems of the UpDown control.

NOTE If you want to scale the text of the form properly, including the captions of components, the items in list boxes, and so on, you should use TrueType fonts exclusively. The system font (MS Sans Serif) doesn't scale well. The font issue is important because the size of many components depends on the text height of their captions, and if the caption does not scale well, the component might not work properly. For this reason, in the Scale example I've used an Arial font.

Exactly the same scaling technique also works in CLX, as you can see by running the QScale example. The only real difference is that I have to replace the UpDown component (and the related Edit box) with a SpinEdit control, as the former is not available in Qt.

Automatic Form Scaling

Instead of playing with the `ScaleBy` method, you can ask Delphi to do the work for you. When Delphi starts, it asks the system for the display configuration and stores the value in the `PixelsPerInch` property of the `Screen` object, a special global object of VCL, available in any application.

PixelsPerInch sounds like it has something to do with the pixel resolution of the screen, but unfortunately, it doesn't. If you change your screen resolution from 640×480 to 800×600 to 1024×768 or even 1600×1280, you will find that Windows reports the same PixelsPerInch value in all cases, unless you change the system font. What PixelsPerInch really refers to is the screen pixel resolution that the currently installed system font was designed for. When a user changes the system font scale, usually to make menus and other text easier to read, the user will expect all applications to honor those settings. An application that does not reflect user desktop preferences will look out of place and, in extreme cases, may be unusable to visually impaired users who rely on very large fonts and high-contrast color schemes.

The most common PixelPerInch values are 96 (small fonts) and 120 (large fonts), but other values are possible. Newer versions of Windows even allow the user to set the system font size to an arbitrary scale. At design time, the PixelsPerInch value of the screen, which is a read-only property, is copied to every form of the application. Delphi then uses the value of PixelsPerInch, if the Scaled property is set to True, to resize the form when the application starts.

As I've already mentioned, both automatic scaling and the scaling performed by the ScaleBy method operate on components by changing the size of the font. The size of each control, in fact, depends on the font it uses. With automatic scaling, the value of the form's PixelsPerInch property (the design-time value) is compared to the current system value (indicated by the corresponding property of the Screen object), and the result is used to change the font of the components on the form. Actually, to improve the accuracy of this code, the final height of the text is compared to the design-time height of the text, and its size is adjusted if they do not match.

Thanks to Delphi automatic support, the same application running on a system with a different system font size automatically scales itself, without any specific code. The application's edit controls will be the correct size to display their text in the user's preferred font size, and the form will be the correct size to contain those controls. Although automatic scaling has problems in some special cases, if you comply with the following rules, you should get good results:

- Set the Scaled property of forms to True. (This is the default.)
- Use only TrueType fonts.
- Use Windows small fonts (96 dpi) on the computer you use to develop the forms.
- Set the AutoScroll property to False, if you want to scale the form and not just the controls inside it. (AutoScroll defaults to True, so don't forget to do this step.)
- Set the form position either near the upper-left corner or in the center of the screen (with the poScreenCenter value) to avoid having an out-of-screen form. Form position is discussed in the next section.

Creating and Closing Forms

Up to now we have ignored the issue of form creation. We know that when the form is created, we receive the `OnCreate` event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in this project's source file:

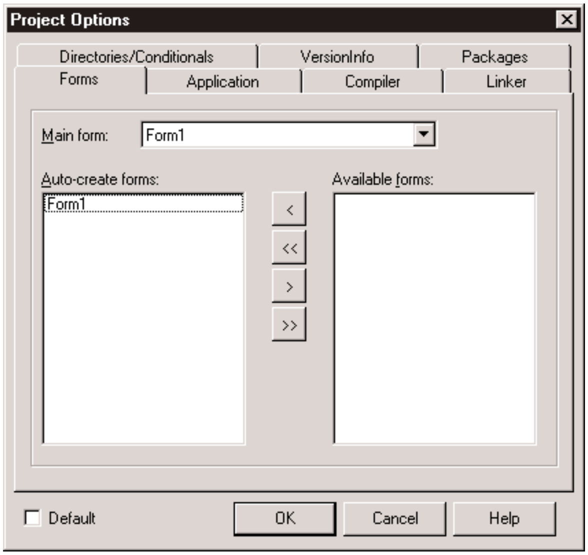
```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box (see Figure 9.11). In this dialog box, you can decide whether the form should be automatically created. If you disable the automatic creation, the project's initialization code becomes the following:

```
begin
  Applications.Initialize;
  Application.Run;
end.
```

FIGURE 9.11:

The Forms page of the Delphi Project Options dialog box



Main form: Form1

Auto-create forms: Form1

Available forms:

If you now run this program, nothing happens. It terminates immediately because no main window is created. So what is the effect of the call to the application's `CreateForm` method? It creates a new instance of the form class passed as the first parameter and assigns it to the variable passed as the second parameter.

Something else happens behind the scenes. When CreateForm is called, if there is currently no main form, the current form is assigned to the application's MainForm property. For this reason, the form indicated as Main Form in the dialog box shown in Figure 9.11 corresponds to the first call to the application's CreateForm method (that is, when several forms are created at start-up).

The same holds for closing the application. Closing the main form terminates the application, regardless of the other forms. If you want to perform this operation from the program's code, simply call the Close method of the main form, as we've done several times in past examples.

TIP You can control the automatic creation of secondary forms by using the Auto Create Forms check box on the Preferences page of the Environment Options dialog box.

Form Creation Events

Regardless of the manual or automatic creation of forms, when a form is created, there are many events you can intercept. Form-creation events are fired in the following order:

1. OnCreate indicates that the form is being created.
2. OnShow indicates that the form is being displayed. Besides main forms, this event happens after you set the Visible property of the form to True or call the Show or ShowModal methods. This event is fired again if the form is hidden and then displayed again.
3. OnActivate indicates that the form becomes the active form within the application. This event is fired every time you move from another form of the application to the current one.
4. Other events, including OnResize and OnPaint, indicate operations always done at start-up but then repeated many times.

As you can see in the list above, every event has a specific role apart from form initialization, except for the OnCreate event, which is guaranteed to be called only once as the form is created.

However, there is an alternative approach to adding initialization code to a form: overriding the constructor. This is usually done as follows:

```
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited Create (AOwner);
  // extra initialization code
end;
```

Before the call to the Create method of the base class, the properties of the form are still not loaded and the internal components are not available. For this reason the standard approach is to call the base class constructor first and then do the custom operations.

Old and New Creation Orders

Now the question is whether these custom operations are executed before or after the OnCreate event is fired. The answer depends on the value of the OldCreateOrder property of the form, introduced in Delphi 4 for backward compatibility with earlier versions of Delphi. By default, for a new project, all of the code in the constructor is executed before the OnCreate event handler. In fact, this event handler is not activated by the base class constructor but by its AfterConstruction method, a sort of constructor introduced for compatibility with C++Builder.

To study the creation order and the potential problems, you can examine the CreatOrd program. This program has an OnCreate event handler, which creates a list box control dynamically. The constructor of the form can access this list box or not, depending on the value of the OldCreateOrder property.

Closing a Form

When you close the form using the Close method or by the usual means (Alt+F4, the system menu, or the Close button), the OnCloseQuery event is called. In this event, you can ask the user to confirm the action, particularly if there is unsaved data in the form. Here is a simple scheme of the code you can write:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg ('Are you sure you want to exit?', mtConfirmation,
      [mbYes, mbNo], 0) = idNo then
    CanClose := False;
end;
```

If OnCloseQuery indicates that the form should still be closed, the OnClose event is called. The third step is to call the OnDestroy event, which is the opposite of the OnCreate event and is generally used to de-allocate objects related to the form and free the corresponding memory.

NOTE To be more precise, the BeforeDestruction method generates an OnDestroy event before the Destroy destructor is called. That is, unless you have set the OldCreateOrder property to True, in which case Delphi uses a different closing sequence.

So what is the use of the intermediate OnClose event? In this method, you have another chance to avoid closing the application, or you can specify alternative “close actions.” The method, in fact, has an Action parameter passed by reference. You can assign the following values to this parameter:

caNone The form is not allowed to close. This corresponds to setting the CanClose parameter of the OnCloseQuery method to False.

caHide The form is not closed, just hidden. This makes sense if there are other forms in the application; otherwise, the program terminates. This is the default for secondary forms, and it’s the reason I had to handle the OnClose event in the previous example to actually close the secondary forms.

caFree The form is closed, freeing its memory, and the application eventually terminates if this was the main form. This is the default action for the main form and the action you should use when you create multiple forms dynamically (if you want to remove the Windows and destroy the corresponding Delphi object as the form closes).

caMinimize The form is not closed but only minimized. This is the default action for MDI child forms.

NOTE When a user shuts down Windows, the OnCloseQuery event is activated, and a program can use it to stop the shut-down process. In this case, the OnClose event is not called even if OnCloseQuery sets the CanClose parameter to True.

Dialog Boxes and Other Secondary Forms

When you write a program, there is really no big difference between a dialog box and another secondary form, aside from the border, the border icons, and similar user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have nonmodal—or *modeless*—dialog boxes. So if you think that dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. We have to consider two different elements:

- The form’s border and its user interface determine whether it looks like a dialog box.
- The use of two different methods (Show or ShowModal) to display the secondary form determines its behavior (modeless or modal).

Adding a Second Form to a Program

To add a second form to an application, you simply click on the New Form button on the Delphi toolbar or use the File ➢ New Form menu command. As an alternative you can select File ➢ New, move to the Forms or Dialogs page, and choose one of the available form templates or form wizards.

If you have two forms in a project, you can use the Select Form or Select Unit button of the Delphi toolbar to navigate through them at design time. You can also choose which form is the main one and which forms should be automatically created at start-up using the Forms page of the Project Options dialog box. This information is reflected in the source code of the project file.

TIP Secondary forms are automatically created in the project source-code file depending on a new Delphi 5 setting, which is the Auto Create Forms check box of the Preferences page of the Environment Options dialog box. Although automatic creation is the simplest and most reliable approach for novice developers and quick-and-dirty projects, I suggest that you disable this check box for any serious development. When your application contains hundreds of forms, you really shouldn't have them all created at application start-up. Create instances of secondary forms when and where you need them, and free them when you're done.

Once you have prepared the secondary form, you can simply set its `Visible` property to True, and both forms will show up as the program starts. In general, the secondary forms of an application are left "invisible" and are then displayed by calling the `Show` method (or setting the `Visible` property at run time). If you use the `Show` function, the second form will be displayed as modeless, so you can move back to the first one while the second is still visible. To close the second form, you might use its system menu or click a button or menu item that calls the `Close` method. As we've just seen, the default close action (see the `OnClose` event) for a secondary form is simply to hide it, so the secondary form is not destroyed when it is closed. It is kept in memory (again, not always the best approach) and is available if you want to show it again.

Creating Secondary Forms at Run Time

Unless you create all the forms when the program starts, you'll need to check whether a form exists and create it if necessary. The simplest case is when you want to create multiple copies of the same form at run time. In the MultiWin/QMultiWin example, I've done this by writing the following code:

```
procedure TForm1.btnMultipleClick(Sender: TObject);
begin
  with TForm3.Create (Application) do
    Show;
end;
```

Every time you click the button, a new copy of the form is created. Notice that I don't use the Form3 global variable, because it doesn't make much sense to assign this variable a new value every time you create a new form object. The important thing, however, is not to refer to the global Form3 object in the code of the form itself or in other portions of the application. The Form3 variable, in fact, will invariably be a pointer to nil. My suggestion, in such a case, is to actually remove it from the unit to avoid any confusion.

TIP

In the code of a form, you should never explicitly refer to the form by using the global variable that Delphi sets up for it. For example, suppose that in the code of TForm3 you refer to Form3.Caption. If you create a second object of the same type (the class TForm3), the expression Form3.Caption will invariably refer to the caption of the form object referenced by the Form3 variable, which might not be the current object executing the code. To avoid this problem, refer to the Caption property in the form's method to indicate the caption of the current form object, and use the Self keyword when you need a specific reference to the object of the current form. To avoid any problem when creating multiple copies of a form, I suggest removing the global form object from the interface portion of the unit declaring the form. This global variable is required only for the automatic form creation.

When you create multiple copies of a form dynamically, remember to destroy each form object as is it closed, by handling the corresponding event:

```
procedure TForm3.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Failing to do so will result in a lot of memory consumption, because all the forms you create (both the windows and the Delphi objects) will be kept in memory and simply hidden from view.

Creating Single-Instance Secondary Forms

Now let us focus on the dynamic creation of a form, in a program that accounts for only one copy of the form at a time. Creating a modal form is quite simple, because the dialog box can be destroyed when it is closed, with code like this:

```
procedure TForm1.btnModalClick(Sender: TObject);
var
  Modal: TForm4;
begin
  Modal := TForm4.Create (Application);
  try
    Modal.ShowModal;
  finally
    Modal.Free;
  end;
end;
```

Because the ShowModal call can raise an exception, you should write it in a finally block to make sure the object will be de-allocated. Usually this block also includes code that initializes the dialog box before displaying it and code that extracts the values set by the user before destroying the form. The final values are read-only if the result of the ShowModal function is mrOK, as we'll see in the next example.

The situation is a little more complex when you want to display only one copy of a modeless form. In fact, you have to create the form, if it is not already available, and then show it:

```
procedure TForm1.btnSingleClick(Sender: TObject);
begin
  if not Assigned (Form2) then
    Form2 := TForm2.Create (Application);
  Form2.Show;
end;
```

With this code, the form is created the first time it is required and then is kept in memory, visible on the screen or hidden from view. To avoid using up memory and system resources unnecessarily, you'll want to destroy the secondary form when it is closed. You can do that by writing a handler for the OnClose event:

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  // important: set pointer to nil!
  Form2 := nil;
end;
```

Notice that after we destroy the form, the global Form2 variable is set to nil. Without this code, closing the form would destroy its object, but the Form2 variable would still refer to the original memory location. At this point, if you try to show the form once more with the btnSingleClick method shown earlier, the if not Assigned test will succeed, as it simply checks whether the Form2 variable is nil. The code fails to create a new object, and the Show method, invoked on a nonexistent object, will result in a system memory error.

As an experiment, you can generate this error by removing the last line of the listing above. As we have seen, the solution is to set the Form2 object to nil when the object is destroyed, so that properly written code will "see" that a new form has to be created before using it. Again, experimenting with the MultiWin/QMultiWin example can prove useful to test various conditions. I haven't illustrated any screens from this example because the forms it displays are quite bare (totally empty except for the main form, which has three buttons).

NOTE Setting the form variable to nil makes sense—and works—if there is to be only one instance of the form present at any given instant. If you want to create multiple copies of a form, you'll have to use other techniques to keep track of them. Also keep in mind that in this case we cannot use the FreeAndNil procedure, because we cannot call Free on Form2. The reason is that we cannot destroy the form before its event handlers have finished executing.

Creating a Dialog Box

I stated earlier in this chapter that a dialog box is not very different from other forms. To build a dialog box instead of a form, you just select the bsDialog value for the `BorderStyle` property. With this simple change, the interface of the form becomes like that of a dialog box, with no system icon, and no Minimize or Maximize boxes. Of course, such a form has the typical thick dialog box border, which is non-resizable.

Once you have built a dialog box form, you can display it as a modal or modeless window using the two usual show methods (`Show` and `ShowModal`). Modal dialog boxes, however, are more common than modeless ones. This is exactly the reverse of forms; modal forms should generally be avoided, because a user won't expect them.

The Dialog Box of the RefList Example

In Chapter 6 we explored the RefList/QRefList program, which used a ListView control to display references to books, magazines, Web sites, and more. In the RefList2 version on the CD (and its QRefLsit2 CLX counterpart), I'll simply add to the basic version of that program a dialog box, used in two different circumstances: adding new items to the list and editing existing items. You can see the form of the dialog box in Figure 9.12 and its textual description in the following listing (detailed because it has many interesting features, so I suggest you read this code with care).

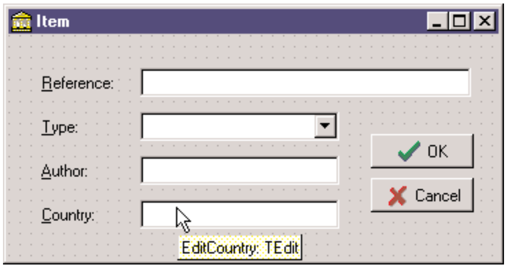


FIGURE 9.12:

The form of the dialog box of the RefList2 example at design time

```
object FormItem: TFormItem
  Caption = 'Item'
  Color = clBtnFace
  Position = poScreenCenter
  object Label1: TLabel
    Caption = '&Reference:'
    FocusControl = EditReference
  end
  object EditReference: TEdit...
  object Label2: TLabel
```

```
    Caption = '&Type:'
    FocusControl = ComboType
  end
  object ComboType: TComboBox
    Style = csDropDownList
    Items.Strings = (
      'Book'
      'CD'
      'Magazine'
      'Mail Address'
      'Web Site')
  end
  object Label3: TLabel
    Caption = '&Author:'
    FocusControl = EditAuthor
  end
  object EditAuthor: TEdit...
  object Label4: TLabel
    Caption = '&Country:'
    FocusControl = EditCountry
  end
  object EditCountry: TEdit...
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

TIP The items of the combo box in this dialog describe the available images of the image list so that a user can select the type of the item and the system will show the corresponding glyph. An even better option would have been to show those glyphs in a graphical combo box, along with their descriptions.

As I mentioned, this dialog box is used in two different cases. The first takes place as the user selects File ➢ Add Items from the menu:

```
procedure TForm1.AddItems1Click(Sender: TObject);
var
  NewItem: TListItem;
begin
  FormItem.Caption := 'New Item';
  FormItem.Clear;
  if FormItem.ShowModal = mrOK then
  begin
```

```
    NewItem := ListView1.Items.Add;
    NewItem.Caption := FormItem.EditReference.Text;
    NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
    NewItem.SubItems.Add (FormItem.EditAuthor.Text);
    NewItem.SubItems.Add (FormItem.EditCountry.Text);
  end;
end;
```

Besides setting the proper caption of the form, this procedure needs to initialize the dialog box, as we are entering a brand-new value. If the user clicks OK, however, the program adds a new item to the list view and sets all its values. To empty the edit boxes of the dialog, the program calls the custom `Clear` method, which resets the text of each edit box control:

```
procedure TFormItem.Clear;
var
  I: Integer;
begin
  // clear each edit box
  for I := 0 to ControlCount - 1 do
    if Controls [I] is TEdit then
      TEdit (Controls[I]).Text := '';
end;
```

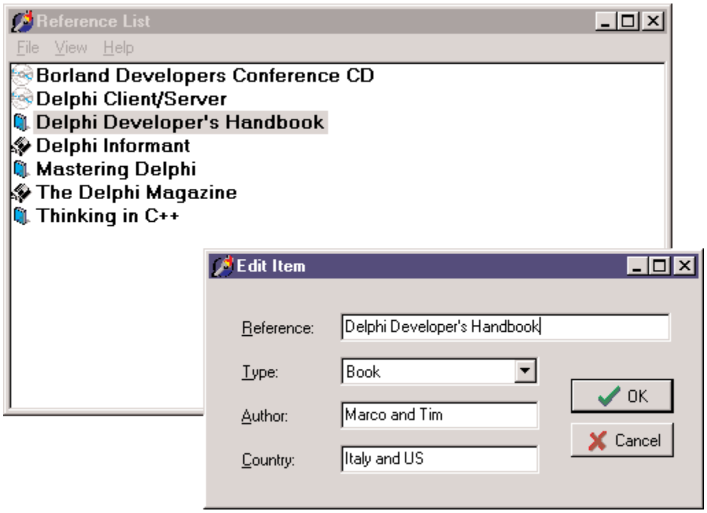
Editing an existing item requires a slightly different approach. First, the current values are moved to the dialog box before it is displayed. Second, if the user clicks OK, the program modifies the current list item instead of creating a new one. Here is the code:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
begin
  if ListView1.Selected <> nil then
  begin
    // dialog initialization
    FormItem.Caption := 'Edit Item';
    FormItem.EditReference.Text := ListView1.Selected.Caption;
    FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
    FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
    FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

    // show it
    if FormItem.ShowModal = mrOK then
    begin
      // read the new values
      ListView1.Selected.Caption := FormItem.EditReference.Text;
      ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
      ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
      ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
    end;
  end;
end;
```

You can see the effect of this code in Figure 9.13. Notice that the code used to read the value of a new item or modified one is similar. In general, you should try to avoid this type of duplicated code and possibly place the shared code statements in a method added to the dialog box. In this case, the method could receive as parameter a `TListItem` object and copy the proper values into it.

FIGURE 9.13:

The dialog box of the RefList2 example used in edit mode



Reference: Delphi Developer's Handbook

Type: Book

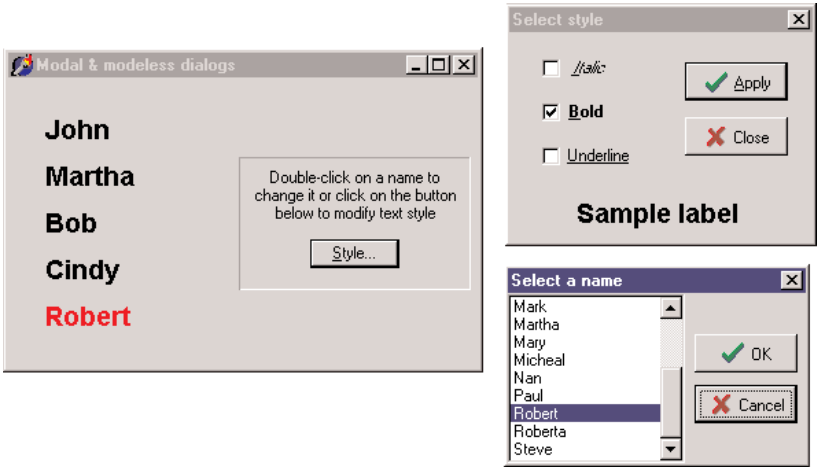
Author: Marco and Tim

Country: Italy and US

NOTE What happens internally when the user clicks the OK or Cancel button of the dialog box? A modal dialog box is closed by setting its `ModalResult` property, and it returns the value of this property. You can indicate the return value by setting the `ModalResult` property of the button. When the user clicks on the button, its `ModalResult` value is copied to the form, which closes the form and returns the value as the result of the `ShowModal` function.

A Modeless Dialog Box

The second example of dialog boxes shows a more complex modal dialog box that uses the standard approach as well as a modeless dialog box. The main form of the DlgApply example (and of the identical CLX-based QDlgApply demo) has five labels with names, as you can see in Figure 9.14 and by viewing the source code on the companion CD.



Double-click on a name to change it or click on the button below to modify text style

FIGURE 9.14: The three forms (a main form and two dialog boxes) of the DlgApply example at run time

If the user clicks a name, its color turns to red; if the user double-clicks it, the program displays a modal dialog box with a list of names to choose from. If the user clicks the Style button, a modeless dialog box appears, allowing the user to change the font style of the main form's labels. The five labels of the main form are connected to two methods, one for the `OnClick` event and the second for the `OnDoubleClick` event. The first method turns the last label a user has clicked to red, resetting to black all the others (which have the `Tag` property set to 1, as a sort of group index). Notice that the same method is associated with all of the labels:

```
procedure TForm1.LabelClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount - 1 do
   if (Components[I] is TLabel) and (Components[I].Tag = 1) then
     TLabel (Components[I]).Font.Color := clBlack;
  // set the color of the clicked label to red
  (Sender as TLabel).Font.Color := clRed;
end;
```

The second method common to all of the labels is the handler of the `OnDoubleClick` event. The `LabelDoubleClick` method selects the `Caption` of the current label (indicated by the `Sender` parameter) in the list box of the dialog and then shows the modal dialog box. If the user closes the dialog box by clicking OK and an item of the list is selected, the selection is copied back to the label's caption:

```
procedure TForm1.LabelDoubleClick(Sender: TObject);
begin
  with ListDial.Listbox1 do
  begin
    // select the current name in the list box
    ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
    // show the modal dialog box, checking the return value
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
      // copy the selected item to the label
      (Sender as TLabel).Caption := Items [ItemIndex];
  end;
end;
```

TIP Notice that all the code used to customize the modal dialog box is in the LabelDoubleClick method of the main form. The form of this dialog box has no added code.

The modeless dialog box, by contrast, has a lot of coding behind it. The main form simply displays the dialog box when the Style button is clicked (notice that the button caption ends with three dots to indicate that it leads to a dialog box), by calling its Show method. You can see the dialog box running in Figure 9.14 above.

Two buttons, Apply and Close, replace the OK and Cancel buttons in a modeless dialog box. (The fastest way to obtain these buttons is to select the bkOK or bkCancel value for the Kind property and then edit the Caption.) At times, you may see a Cancel button that works as a Close button, but the OK button in a modeless dialog box usually has no meaning. Instead, there might be one or more buttons that perform specific actions on the main window, such as Apply, Change Style, Replace, Delete, and so on.

If the user clicks one of the check boxes of this modeless dialog box, the style of the sample label's text at the bottom changes accordingly. You accomplish this by adding or removing the specific flag that indicates the style, as in the following OnClick event handler:

```
procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);
begin
  if ItalicCheckBox.Checked then
    LabelSample.Font.Style := LabelSample.Font.Style + [fsItalic]
  else
    LabelSample.Font.Style := LabelSample.Font.Style - [fsItalic];
end;
```

When the user selects the Apply button, the program copies the style of the sample label to each of the form's labels, rather than considering the values of the check boxes:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
begin
  Form1.Label1.Font.Style := LabelSample.Font.Style;
  Form1.Label2.Font.Style := LabelSample.Font.Style;
  ...
```

As an alternative, instead of referring to each label directly, you can look for it by calling the FindComponent method of the form, passing the label name as a parameter, and then casting the result to the TLabel type. The advantage of this approach is that we can create the names of the various labels with a for loop:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    (Form1.FindComponent ('Label' + IntToStr (I)) as TLabel).Font.Style :=
      LabelSample.Font.Style;
end;
```

TIP The ApplyBitBtnClick method could also be written by scanning the Controls array in a loop, as I've already done in other examples. I decided to use the FindComponent method, instead, to demonstrate a different technique.

This second version of the code is certainly slower, because it has more operations to do, but you won't notice the difference, because it is very fast anyway. Of course, this second approach is also more flexible; if you add a new label, you only need to fix the higher limit of the for loop, provided all the labels have consecutive numbers. Notice that when the user clicks the Apply button, the dialog box does not close. Only the Close button has this effect. Consider also that this dialog box needs no initialization code because the form is not destroyed, and its components maintain their status each time the dialog box is displayed.

Predefined Dialog Boxes

Besides building your own dialog boxes, Delphi allows you to use some default dialog boxes of various kinds. Some are predefined by Windows; others are simple dialog boxes (such as message boxes) displayed by a Delphi routine. The Delphi Component Palette contains a page of dialog box components. Each of these dialog boxes—known as *Windows common dialogs*—is defined in the system library ComDlg32.DLL.

Windows Common Dialogs

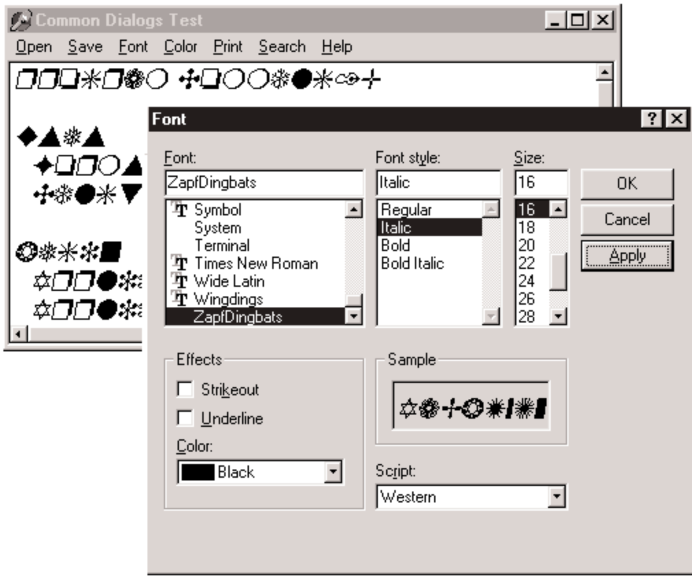
I have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the Execute method, returning a Boolean value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, I've built the CommDlg test program.

What I want to do is simply highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The Open Dialog Component can be customized by setting different file extensions filters, using the `Filter` property, which has a handy editor and can be assigned directly with a string like `Text File (*.txt)|*.txt`. Another handy feature is to let the dialog check whether the extension of the selected file matches the default extension, by checking the ofExtensionDifferent flag of the `Options` property after executing the dialog. Finally, this dialog allows multiple selections by setting its ofAllowMultiSelect option. In this case you can get the list of the selected files by looking at the `Files` string list property.
- The SaveDialog component is used in similar ways and has similar properties, although you cannot select multiple files, of course.
- The OpenPictureDialog and SavePictureDialog components provide similar features but have a customized form, which shows a preview of an image. Of course, it makes sense to use them only for opening or saving graphical files.
- The FontDialog component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (WYSIWYG), or only TrueType fonts. You can show or hide the portion related to the special effects, and obtain other different versions by setting its `Options` property. You can also activate an Apply button simply by providing an event handler for its `OnApply` event and using the fdApplyButton option. A Font dialog box with an Apply button (see Figure 9.15) behaves almost like a modeless dialog box (but isn't one).

FIGURE 9.15:

The Font selection dialog box with an Apply button



Font: ZapfDingbats
Font style: Italic
Size: 16
☐ Strikeout
☐ Underline
Color: Black
Script: Western

- The ColorDialog component is used with different options, to show the dialog fully open at first or to prevent it from opening fully. These settings are the cdFullOpen or cdPreventFullOpen values of the `Options` property.
- The Find and Replace dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the CommDlg example. The custom code is connected to the buttons of the two dialog boxes by providing the `OnFind` and `OnReplace` events.

NOTE Qt offers a similar set of predefined dialog boxes, only the set of options is often more limited. I've created the QCommDlg version of the example you can use to experiment with these settings. The CLX program has fewer menu items, as some of the options are not available and there are other minimal changes in the source code.

A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. There are many Delphi procedures and functions you can use to display simple dialog boxes:

- The `MessageDlg` function shows a customizable message box, with one or more buttons and usually a bitmap. The `MessageDlgPos` function is similar to the `MessageDlg` function, but the message box is displayed in a given position, not in the center of the screen.
- The `ShowMessage` procedure displays a simpler message box, with the application name as the caption and just an OK button. The `ShowMessagePos` procedure does the same, but you also indicate the position of the message box. The `ShowMessageFmt` procedure is a variation of `ShowMessage`, which has the same parameters as the `Format` function. It corresponds to calling `Format` inside a call to `ShowMessage`.
- The `MessageBox` method of the `Application` object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the `MessageBox` function of the Windows API, which passes as a main window parameter the handle of the `Application` object. This handle is required to make the message box behave like a modal window.
- The `InputBox` function asks the user to input a string. You provide the caption, the query, and a default string. The `InputQuery` function asks the user to input a string, too. The only difference between this and the `InputBox` function is in the syntax. The `InputQuery` function has a Boolean return value that indicates whether the user has clicked OK or Cancel.

To demonstrate some of the message boxes available in Delphi, I've written another sample program, with a similar approach to the preceding CommDlg example. In the MBParade example, you have a high number of choices (radio buttons, check boxes, edit boxes, and spin edit controls) to set before you click one of the buttons that displays a message box. The similar QMbParade example misses only the possibility of the help button, not available in the CLX message boxes.

About Boxes and Splash Screens

Applications usually have an About box, where you can display information, such as the version of the product, a copyright notice, and so on. The simplest way to build an About box is to use the `MessageDlg` function. With this method, you can show only a limited amount of text and no special graphics.

Therefore, the usual method for creating an About box is to use a dialog box, such as the one generated with one of the Delphi default templates. In this about box you might want to add some code to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

Building a Splash Screen

Another typical technique used in applications is to display an initial screen before the main form is shown. This makes the application seem more responsive, because you show something to the user while the program is loading, but it also makes a nice visual effect. Sometimes, this same window is displayed as the application's About box.

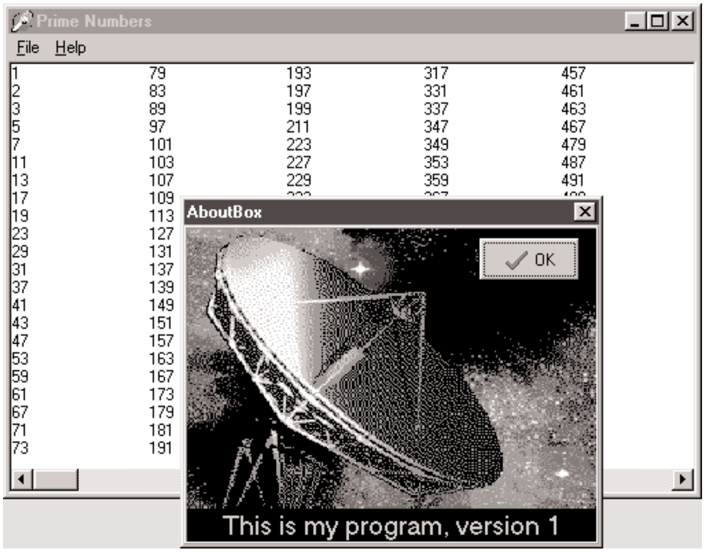
For an example in which a splash screen is particularly useful, I've built a program displaying a list box filled with prime numbers. The prime numbers are computed on program startup, so that they are displayed as soon as the form becomes visible:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 30000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
end;
```

This method calls an `IsPrime` function I've added to the program. This function, which you can find in the source code, computes prime numbers in a terribly slow way; but I needed a slow form creation to demonstrate my point. The numbers are added to a list box that covers the full client area of the form and allows multiple columns to be displayed, as you can see in Figure 9.16.

FIGURE 9.16:

The main form of the Splash example, with the About box activated from the menu



```
1     79    193   317   457
2     83    197   331   461
3     89    199   337   463
5     97    211   347   467
7     101   223   349   479
11    103   227   353   487
13    107   229   359   491
17    109
19    113
23    127
29    131
31    137
37    139
41    149
43    151
47    157
53    163
59    167
61    173
67    179
71    181
73    191
```

This is my program, version 1

There are three versions of the Splash program (plus the three corresponding CLX versions). As you can see by running the Splash0 example, the problem with this program is that the initial operation, which takes place in the `FormCreate` method, takes a lot of time. When you start the program, it takes several seconds to display the main form. If your computer is very fast or very slow, you can change the upper limit of the `for` loop of the `FormCreate` method to make the program faster or slower.

This program has a simple dialog box with an image component, a simple caption, and a bitmap button, all placed inside a panel taking up the whole surface of the About box. This form is displayed when you select the Help ➢ About menu item. But what we really want is to display this About box while the program starts. You can see this effect by running the Splash1 and Splash2 examples, which show a splash screen using two different techniques.

First of all, I've added a method to the `TAboutBox` class. This method, called `MakeSplash`, changes some properties of the form to make it suitable for a splash form. Basically it removes the border and caption, hides the OK button, makes the border of the panel thick (to replace the border of the form), and then shows the form, repainting it immediately:

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
  Panel1.BorderWidth := 3;
```

```
    Show;
    Update;
  end;
```

This method is called after creating the form in the project file of the Splash1 example. This code is executed before creating the other forms (in this case only the main form), and the splash screen is then removed before running the application. These operations take place within a `try/finally` block. Here is the source code of the main block of the project file for the Splash2 example:

```
var
  SplashAbout: TAboutBox;

begin
  Application.Initialize;

  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  try
    SplashAbout.MakeSplash;
    // standard code...
    Application.CreateForm(TForm1, Form1);
    // get rid of the splash form
    SplashAbout.Close;
  finally
    SplashAbout.Free;
  end;

  Application.Run;
end.
```

This approach makes sense only if your application's main form takes a while to create, to execute its startup code (as in this case), or to open database tables. Notice that the splash screen is the first form created, but because the program doesn't use the `CreateForm` method of the `Application` object, this doesn't become the main form of the application. In this case, in fact, closing the splash screen would terminate the program!

An alternative approach is to keep the splash form on the screen a little longer and use a timer to get rid of it after a while. I've implemented this second technique in the Splash2 example. This example also uses a different approach for creating the splash form: instead of creating it in the project source code, it creates the form at the very beginning of the `FormCreate` method of the main form.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  SplashAbout: TAboutBox;
```

```
begin
  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  SplashAbout.MakeSplash;
  // standard code...
  for I := 1 to 30000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
  // get rid of the splash form, after a while
  SplashAbout.Timer1.Enabled := True;
end;
```

The timer is enabled just before terminating the method. After its interval has elapsed (in the example, 3 seconds) the `OnTimer` event is activated, and the splash form handles it by closing and destroying itself:

```
procedure TAboutBox.Timer1Timer(Sender: TObject);
begin
  Close;
  Release;
end;
```

NOTE The `Release` method of a form is similar to the `Free` method of objects, only the destruction of the form is delayed until all event handlers have completed execution. Using `Free` inside a form might cause an access violation, as the internal code, which fired the event handler, might refer again to the form object.

There is one more thing to fix. The main form will be displayed later and in front of the splash form, unless you make this a top-most form. For this reason I've added one line to the `MakeSplash` method of the About box in the Splash2 example:

```
FormStyle := fsStayOnTop;
```

What's Next?

In this chapter we've explored some important form properties. Now you know how to handle the size and position of a form, how to resize it, and how to get mouse input and paint over it. You know more about dialog boxes, modal forms, predefined dialogs, splash screens, and many other techniques, including the funny effect of alpha blending. Understanding the details of working with forms is critical to a proper use of Delphi, particularly for building complex applications (unless, of course, you're building services or Web applications with no user interface).

In the next chapter we'll continue by exploring the overall structure of a Delphi application, with coverage of the role of two global objects, `Application` and `Screen`. I'll also discuss MDI development as you learn some more advanced features of forms, such as visual form inheritance. I'll also discuss frames, visual component containers similar to forms.

In this chapter, I've also provided a short introduction to direct painting and to the use of the `TCanvas` class. More about graphics in Delphi forms can also be found in the bonus chapter "Graphics in Delphi" on the companion CD.

CHAPTER 10

The Architecture of Delphi Applications

- The *Application* and *Screen* global objects
- Messages and multitasking in Windows
- Finding the previous instance of an application
- MDI applications
- Visual form inheritance
- Frames
- Base forms and interfaces

Although together we've built Delphi applications since the beginning of the book, we've never really focused on the structure and the architecture of an application built with Delphi's class library. For example, there hasn't been much coverage about the global `Application` object, about techniques for keeping tracks of the forms we've created, about the flow of messages in the system, and other such elements.

In the last chapter you saw how to create applications with multiple forms and dialog boxes, but we haven't discussed how these forms can be related one to the other, how can you share similar features of forms, and how you can operate on multiple similar forms in a coherent way. All of this is the ambitious goal of this chapter, which covers both basic and advanced techniques, including visual form inheritance, the use of frames, and MDI development, but also the use of interfaces for building complex hierarchies of form classes.

The *Application* Object

I've already mentioned the `Application` global object on multiple occasions, but as in this chapter we are focusing on the structure of Delphi applications, it is time to delve into some more details of this global object and its corresponding class. `Application` is a global object of the `TApplication` class, defined in the Forms unit and created in the Controls unit.

The `TApplication` class is a component, but you cannot use it at design time. Some of its properties can be directly set in the Application page of the Project Options dialog box; others must be assigned in code.

To handle its events, instead, Delphi includes a handy ApplicationEvents component. Besides allowing you to assign handlers at design time, the advantage of this component is that it allows for multiple handlers. If you simply place two instances of the ApplicationEvents component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple ApplicationEvents components can chain the handlers.

Some of these application-wide events, including `OnActivate`, `OnDeactivate`, `OnMinimize`, and `OnRestore`, allow you to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as in `OnActionExecute`, `OnActionUpdate`, `OnHelp`, `OnHint`, `OnShortCut`, and `OnShowHint`. Finally, there is the `OnException` global exception handler we used in Chapter 3, the `OnIdle` event used for background computing, and the `OnMessage` event, which fires whenever a message is posted to any of the windows or windowed controls of the application.

Although its class inherits directly from `TComponent`, the `Application` object has a window associated with it. The application window is hidden from sight but appears on the Taskbar. This is why Delphi names the window *Form1* and the corresponding Taskbar icon *Project1*.

The window related to the Application object—the application window—serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this invisible owner window, for example, is fundamental when the application is activated. In fact, when the windows of your program are behind those of other programs, clicking one window in your application will bring all of that application's windows to the front. In other words, the unseen application window is used to connect the various forms of the application. Actually the application window is not *hidden*, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.

TIP In Windows, the Minimize and Maximize operations are associated by default with system sounds and a visual animated effect. Applications built with Delphi (starting with version 5) produce the sound and display the visual effect by default.

When you create a new, blank application, Delphi generates a code for the project file, which includes the following:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

As you can see in this standard code, the Application object can create forms, setting the first one as the MainForm (one of the Application properties) and closing the entire application when this main form is destroyed. Moreover, it contains the Windows message loop (started by the Run method) that delivers the system messages to the proper windows of the application. A message loop is required by any Windows application, but you don't need to write one in Delphi because the Application object provides a default one.

If this is the main role of the Application object, it manages few other interesting areas as well:

- Hints (discussed at the end of Chapter 7)
- The help system, which in Delphi 6 includes the ability to define the type of help viewer (something not covered in detail in this book)
- Application activation, minimize, and restore
- A global exceptions handler, as discussed in Chapter 3 in the ErrorLog example
- General application information, including the MainForm, executable file name and path (ExeName), the Icon, and the Title displayed in the Windows taskbar and when you scan the running applications with the Alt+Tab keys

TIP

To avoid a discrepancy between the two titles, you can change the application's title at design time. As an alternative, at run time, you can copy the form's caption to the title of the application with this code: `Application.Title := Form1.Caption`.

In most applications, you don't care about the application window, apart from setting its `Title` and icon and handling some of its events. There are some simple operations you can do anyway. Setting the `ShowMainForm` property to False in the project source code indicates that the main form should not be displayed at startup. Inside a program, instead, you can use the `MainForm` property of the `Application` object to access the main form, which is the first form created in the program.

Displaying the Application Window

There is no better proof that a window indeed exists for the `Application` object than to display it. Actually, we don't need to show it—we just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. We can perform these operations by using Windows API functions on the window indicated by the `Handle` property of the `Application` object:

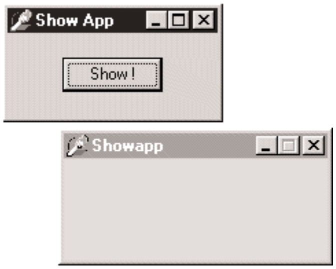
```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldStyle: Integer;
begin
  // add border and caption to the app window
  OldStyle := GetWindowLong (Application.Handle, gwl_Style);
  SetWindowLong (Application.Handle, gwl_Style,
    OldStyle or ws_ThickFrame or ws_Caption);
  // set the size of the app window
  SetWindowPos (Application.Handle, 0, 0, 0, 200, 100,
    swp_NoMove or swp_NoZOrder);
end;
```

The two `GetWindowLong` and `SetWindowLong` API functions are used to access the system information related to the window. In this case, we are using the `gwl_Style` parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. The code above gets the current styles and adds (using an `or` statement) a standard border and a caption to the form. As we'll see later in this chapter, you seldom need to use these low-level API functions in Delphi, because there are properties of the `TForm` class that have the same effect. We need this code here because the application window is not a form.

Executing this code displays the project window, as you can see in Figure 10.1. Although there's no need to implement something like this in your own programs, running this program will reveal the relation between the application window and the main window of a Delphi program. This is a very important starting point if you want to understand the internal structure of Delphi applications.

FIGURE 10.1:

The hidden application window revealed by the ShowApp program



The Application System Menu

Unless you write a very odd program like the example we've just looked at, users will only see the application window in the Taskbar. There, they can activate the window's system menu by right-clicking it. As I mentioned in the SysMenu example in Chapter 6, when discussing the system menu, an application's menu is not the same as that of the main form. In that example, I added custom items to the system menu of the main form. Now in the SysMenu2 example, I want to customize the system menu of the application window in the Taskbar.

First we have to add the new items to the system menu of the application window when the program starts. Here is the updated code of the `FormCreate` method:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // add a separator and a menu item to the system menu
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_STRING, idSysAbout,
    '&About...');
  // add the same items to the application system menu
  AppendMenu (GetSystemMenu (Application.Handle, FALSE), MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Application.Handle, FALSE), MF_STRING, idSysAbout,
    '&About...');
end;
```

The first part of the code adds the new separator and item to the system menu of the main form. The other two calls add the same two items to the application's system menu, simply by referring to `Application.Handle`. This is enough to display the updated system menu, as you can see by running this program. The next step is to handle the selection of the new menu item.

To handle form messages, we can simply write new event handlers or message-handling methods. We cannot do the same with the application window, simply because inheriting from the `TApplication` class is quite a complex issue. Most of the time we can just handle the OnMessage event of this class, which is activated for every message the application retrieves from the message queue.

To handle the OnMessage event of the global Application object, simply add an ApplicationEvents component to the main form, and define a handler for the OnMessage event of this component. In this case, we only need to handle the wm_SysCommand message, and we only need to do that if the wParam parameter indicates that the user has selected the menu item we've just added, idSysAbout:

```
procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
  var Handled: Boolean);
begin
  if (Msg.Message = wm_SysCommand) and (Msg.wParam = idSysAbout) then
  begin
    ShowMessage ('Mastering Delphi: SysMenu2 example');
    Handled := True;
  end;
end;
```

This method is very similar to the one used to handle the corresponding system menu item of the main form:

```
procedure WMSysCommand (var Msg: TWMSysCommand);
  message wm_SysCommand;
...
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  // handle a specific command
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu2 example');
  inherited;
end;
```

Activating Applications and Forms

To show how the activation of forms and applications works, I've written a simple, self-explanatory example, available on the companion CD, called ActivApp. This example has two forms. Each form has a Label component (LabelForm) used to display the status of the form. The program uses text and color for this, as the handlers of the OnActivate and OnDeactivate events of the first form demonstrate:

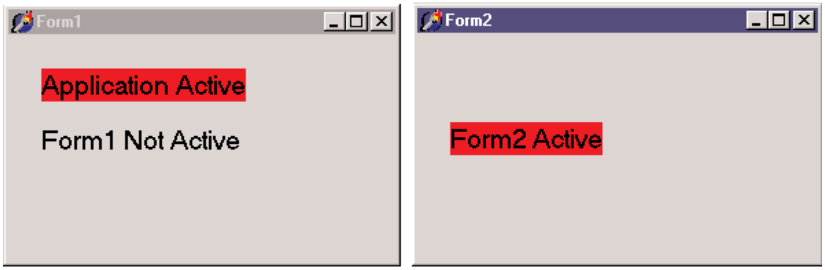
```
procedure TForm1.FormActivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Active';
  LabelForm.Color := clRed;
end;

procedure TForm1.FormDeactivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Not Active';
  LabelForm.Color := clBtnFace;
end;
```

The second form has a similar label and similar code. The main form also displays the status of the entire application. It uses an ApplicationEvents component to handle the `OnActivate` and `OnDeactivate` events of the `Application` object. These two event handlers are similar to the two listed previously, with the only difference being that they modify the text and color of a second label of the form.

If you try running this program, you'll see whether this application is the active one and, if so, which of its forms is the active one. By looking at the output (see Figure 10.2) and listening for the beep, you can understand how each of the activation events is triggered by Delphi. Run this program and play with it for a while to understand how it works. We'll get back to other events related to the activation of forms in a while.

FIGURE 10.2:

The ActivApp example shows whether the application is active and which of the application's forms is active.



Tracking Forms with the *Screen* Object

We have already explored some of the properties and events of the `Application` object. Other interesting global information about an application is available through the `Screen` object, whose base class is `TScreen`. This object holds information about the system display (the screen size and the screen fonts) and also about the current set of forms in a running application. For example, you can display the screen size and the list of fonts by writing:

```
Label1.Caption := IntToStr (Screen.Width) + 'x' + IntToStr (Screen.Height);
ListBox1.Items := Screen.Fonts;
```

`TScreen` also reports the number and resolution of monitors in a multimonitor system. What I want to focus on now, however, is the list of forms held by the `Forms` property of the `Screen` object, the top-most form indicated by the `ActiveForm` property, and the related `OnActiveFormChange` event. Note that the forms the `Screen` object references are the forms of the application and not those of the system.

These features are demonstrated by the Screen example on the CD, which maintains a list of the current forms in a list box. This list must be updated each time a new form is created, an existing form is destroyed, or the active form of the program changes. To see how this works, you can create secondary forms by clicking the button labeled New:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
var
  NewForm: TSecondForm;
begin
  // create a new form, set its caption, and run it
  NewForm := TSecondForm.Create (Self);
  Inc (nForms);
  NewForm.Caption := 'Second ' + IntToStr (nForms);
  NewForm.Show;
end;
```

One of the key portions of the program is the `OnCreate` event handler of the form, which fills the list a first time and then connects a handler to the `OnActiveFormChange` event:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FillFormsList (Self);
  // set the secondary forms counter to 0
  nForms := 0;
  // set an event handler on the screen object
  Screen.OnActiveFormChange := FillFormsList;
end;
```

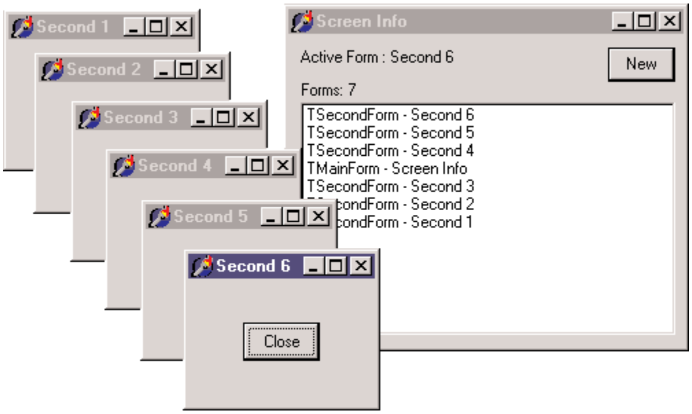
The code used to fill the Forms list box is inside a second procedure, `FillFormsList`, which is also installed as an event handler for the `OnActiveFormChange` event of the `Screen` object:

```
procedure TMainForm.FillFormsList (Sender: TObject);
var
  I: Integer;
begin
  // skip code in destruction phase
  if Assigned (FormsListBox) then
  begin
    FormsLabel.Caption := 'Forms: ' + IntToStr (Screen.FormCount);
    FormsListBox.Clear;
    // write class name and form title to the list box
    for I := 0 to Screen.FormCount - 1 do
      FormsListBox.Items.Add (Screen.Forms[I].ClassName + ' - ' +
        Screen.Forms[I].Caption);
    ActiveLabel.Caption := 'Active Form : ' + Screen.ActiveForm.Caption;
  end;
end;
```

WARNING It is very important not to execute this code while the main form is being destroyed. As an alternative to testing for the listbox not to be set to `nil`, you could as well test the form's `ComponentState` for the csDestroying flag. Another approach would be to remove the `OnActiveFormChange` event handler before exiting the application; that is, handle the `OnClose` event of the main form and assign `nil` to `Screen.OnActiveFormChange`.

The `FillFormsList` method fills the list box and sets a value for the two labels above it to show the number of forms and the name of the active one. When you click the New button, the program creates an instance of the secondary form, gives it a new title, and displays it. The Forms list box is updated automatically because of the handler we have installed for the `OnActiveFormChange` event. Figure 10.3 shows the output of this program when some secondary windows have been created.

FIGURE 10.3: The output of the Screen example with some secondary forms



Active Form : Second 6

Forms: 7

```
TSecondForm - Second 6
TSecondForm - Second 5
TSecondForm - Second 4
TMainForm - Screen Info
TSecondForm - Second 3
condForm - Second 2
condForm - Second 1
```

TIP The program always updates the text of the `ActiveLabel` above the list box to show the currently active form, which is always the same as the first one in the list box.

The secondary forms each have a Close button you can click to remove them. The program handles the `OnClose` event, setting the `Action` parameter to caFree, so that the form is actually destroyed when it is closed. This code closes the form, but it doesn't update the list of the windows properly. The system moves the focus to another window first, firing the event that updates the list, and destroys the old form only after this operation.

The first idea I had to update the windows list properly is to introduce a delay, posting a user-defined Windows message. Because the posted message is queued and not handled immediately, if we send it at the last possible moment of life of the secondary form, the main form will receive it when the other form is destroyed.

The trick is to post the message in the OnDestroy event handler of the secondary form. To accomplish this, we need to refer to the MainForm object, by adding a uses statement in the implementation portion of this unit. I've posted a wm_User message, which is handled by a specific message method of the main form, as shown here:

```
public
  procedure ChildClosed (var Message: TMessage);
    message wm_User;
```

Here is the code for this method:

```
procedure TMainForm.ChildClosed (var Message: TMessage);
begin
  FillFormsList (Self);
end;
```

The problem here is that if you close the main window before closing the secondary forms, the main form exists, but its code cannot be executed anymore. To avoid another system error (an Access Violation Fault), you need to post the message only if the main form is not closing. But how do you know that? One way is to add a flag to the TMainForm class and change its value when the main form is closing, so that you can test the flag from the code of the secondary window.

This is a good solution—so good that the VCL already provides something similar. There is a barely documented ComponentState property. It is a Pascal set that includes (among other flags) a csDestroying flag, which is set when the form is closing. Therefore, we can write the following code:

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

With this code, the list box always lists all of the forms in the application. Note that you need to disable the automatic creation of the secondary form by using the Forms page of the Project Options dialog box.

After giving it some thought, however, I found an alternative and much more Delphi-oriented solution. Every time a component is destroyed, it tells its owner about the event by calling the Notification method defined in the TComponent class. Because the secondary forms are owned by the main one, as specified in the code of the NewButtonClick method, we can override this method and simplify the code. In the form class, simply write

```
protected
  procedure Notification(AComponent: TComponent;
    Operation: TOperation); override;
```

Here is the code of the method:

```
procedure TMainForm.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and Showing and (AComponent is TForm) then
    FillFormsList;
end;
```

You'll find the complete code of this version in the Screen2 directory on the CD.

NOTE In case the secondary forms were not owned by the main one, we could have used the `FreeNotification` method to get the secondary form to notify the main form when they are destroyed. `FreeNotification` receives as parameter the component to notify when the current component is destroyed. The effect is a call to the `Notification` method coming from a component other than the owned ones. `FreeNotification` is generally used by component writers to safely connect components on different forms or data modules.

The last feature I've added to both versions of the program is a simple one. When you click an item in the list box, the corresponding form is activated, using the `BringToFront` method:

```
procedure TMainForm.FormsListBoxClick(Sender: TObject);
begin
  Screen.Forms [FormsListBox.ItemIndex].BringToFront;
end;
```

Nice—well, almost nice. If you click the list box of an inactive form, the main form is activated first, and the list box is rearranged, so you might end up selecting a different form than you were expecting. If you experiment with the program, you'll soon realize what I mean. This minor glitch in the program is an example of the risks you face when you dynamically update some information and let the user work on it at the same time.

Events, Messages, and Multitasking in Windows

To understand how Windows applications work internally, we need to spend a minute discussing how multitasking is supported in this environment. We also need to understand the role of timers (and the Timer component) and of background (or *idle*) computing.

In short, we need to delve deeper into the event-driven structure of Windows and its multitasking support. Because this is a book about *Delphi* programming, I won't discuss this topic in detail, but I will provide an overview for readers who have limited experience with Windows API programming.

Event-Driven Programming

The basic idea behind event-driven programming is that specific events determine the control flow of the application. A program spends most of its time waiting for these events and provides code to respond to them. For example, when a user clicks one of the mouse buttons, an event occurs. A message describing this event is sent to the window currently under the mouse cursor. The program code that responds to events for that window will receive the event, process it, and respond accordingly. When the program has finished responding to the event, it returns to a waiting or “idle” state.

As this explanation shows, events are serialized; each event is handled only after the previous one is completed. When an application is executing event-handling code (that is, when it is not waiting for an event), other events for that application have to wait in a message queue reserved for that application (unless the application uses multiple threads). When an application has responded to a message and returned to a waiting state, it becomes the last in the list of programs waiting to handle additional messages. In every version of Win32 (9*x*, NT, Me, and 2000), after a fixed amount of time has elapsed, the system interrupts the current application and immediately gives control to the next one in the list. The first program is resumed only after each application has had a turn. This is called preemptive multitasking.

So, an application performing a time-consuming operation in an event handler doesn’t prevent the system from working properly, but is generally unable even to repaint its own windows properly, with a very nasty effect. If you’ve never experienced this problem, try for yourself: Write a time-consuming loop executed when a button is pressed, and try to move the form or move another window on top of it. The effect is really annoying. Now try adding the call `Application.ProcessMessages` within the loop, and you’ll see that the operation becomes much slower, but the form will be immediately refreshed.

If an application has responded to its events and is waiting for its turn to process messages, it has no chance to regain control until it receives another message (unless it uses multithreading). This is a reason to use timers, a system component that will send a message to your application every time a time interval elapses.

One final note—when you think about events, remember that input events (using the mouse or the keyboard) account for only a small percentage of the total message flow in a Windows application. Most of the messages are the system’s internal messages or messages exchanged between different controls and windows. Even a familiar input operation such as clicking a mouse button can result in a huge number of messages, most of which are internal Windows messages. You can test this yourself by using the WinSight utility included in Delphi. In WinSight, choose to view the Message Trace, and select the messages for all of the windows. Select Start, and then perform some normal operations with the mouse. You’ll see hundreds of messages in a few seconds.

Windows Message Delivery

Before looking at some real examples, we need to consider another key element of message handling. Windows has two different ways to send a message to a window:

- The PostMessage API function is used to place a message in the application's message queue. The message will be handled only when the application has a chance to access its message queue (that is, when it receives control from the system), and only after earlier messages have been processed. This is an asynchronous call, since you do not know when the message will actually be received.
- The SendMessage API function is used to execute message-handler code immediately. SendMessage bypasses the application's message queue and sends the message directly to a target window or control. This is a synchronous call. This function even has a return value, which is passed back by the message-handling code. Calling SendMessage is no different than directly calling another method or function of the program.

The difference between these two ways of sending messages is similar to that between mailing a letter, which will reach its destination sooner or later, and sending a fax, which goes immediately to the recipient. Although you will rarely need to use these low-level functions in Delphi, this description should help you determine which one to use if you do need to write this type of code.

Background Processing and Multitasking

Suppose that you need to implement a time-consuming algorithm. If you write the algorithm as a response to an event, your application will be stopped completely during all the time it takes to process that algorithm. To let the user know that something is being processed, you can display the hourglass cursor, but this is not a user-friendly solution. Win32 allows other programs to continue their execution, but the program in question will freeze; it won't even update its own user interface if a repaint is requested. In fact, while the algorithm is executing, the application won't be able to receive and process any other messages, including the paint messages.

The simplest solution to this problem is to call the ProcessMessages method of the Application object many times within the algorithm, usually inside an internal loop. This call stops the execution, allows the program to receive and handle a message, and then resumes execution. The problem with this approach, however, is that while the program is paused to accept messages, the user is free to do any operation and might again click the button or press the keystrokes that started the algorithm. To fix this, you can disable the buttons and commands you don't want the user to select, and you can display the hourglass cursor (which technically doesn't prevent a mouse click event, but it does suggest that the user should wait before doing any other operation). An alternative solution is to split the algorithm into smaller pieces and execute each of them in turn, letting the application respond to pending messages in between processing the pieces. We can use a timer to let the system notify us once a time interval has elapsed. Although you can use timers to implement some form of background computing, this is far from a good solution. A slightly better technique would be to execute each step of the program when the Application object receives the OnIdle event.

The difference between calling ProcessMessages and using the OnIdle events is that by calling ProcessMessages, you will give your code more processing time than with the OnIdle approach. Calling ProcessMessages is a way to let the system perform other operations while your program is computing; using the OnIdle event is a way to let your application perform background tasks when it doesn't have pending requests from the user.

NOTE All these techniques for background computing were necessary in 16-bit Windows days. In Win32, you should generally use secondary threads to perform lengthy or background operations.

Checking for a Previous Instance of an Application

One form of multitasking is the execution of two or more instances of the same application. Any application can generally be executed by a user in more than one instance, and it needs to be able to check for a previous instance already running, in order to disable this default behavior and allow for one instance at most. This section demonstrates several ways of implementing such a check, allowing me to discuss some interesting Windows programming techniques.

Looking for a Copy of the Main Window

To find a copy of the main window of a previous instance, use the FindWindow API function and pass it the name of the window class (the name used to register the form's window type, or WNDCLASS, in the system) and the caption of the window for which you are looking. In a Delphi application, the name of the WNDCLASS window class is the same as the Object Pascal name for the form's class (for example, TForm1). The result of the FindWindow function is either a handle to the window or zero (if no matching window was found).

The main code of your Delphi application should be written so that it will execute only if the FindWindow result is zero:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', nil);
  if Hwnd = 0 then
```

```
    begin
      Application.Initialize;
      Application.CreateForm(TForm1, Form1);
      Application.Run;
    end
    else
      SetForegroundWindow (Hwnd)
  end.
```

To activate the window of the previous instance of the application, you can use the `SetForegroundWindow` function, which works for windows owned by other processes. This call produces its effect only if the window passed as parameter hasn't been minimized. When the main form of a Delphi application is minimized, in fact, it is hidden, and for this reason the activation code has no effect.

Unfortunately, if you run a program that uses the `FindWindow` call just shown from within the Delphi IDE, a window with that caption and class may already exist: the design-time form. Thus, the program won't start even once. However, it will run if you close the form and its corresponding source code file (closing only the form, in fact, simply hides the window), or if you close the project and run the program from the Windows Explorer.

Using a Mutex

A completely different approach is to use a *mutex*, or mutual exclusion object. This is a typical Win32 approach, commonly used for synchronizing threads, as we'll see later in this chapter. Here we are going to use a mutex for synchronizing two different applications, or (to be more precise) two instances of the same application.

Once an application has created a mutex with a given name, it can test whether this object is already owned by another application, calling the `WaitForSingleObject` Windows API function. If the mutex has no owner, the application calling this function becomes the owner. If the mutex is already owned, the application waits until the time-out (the second parameter of the function) elapses. It then returns an error code.

To implement this technique, you can use the following project source code, which you'll find in the OneCopy example:

```
var
  hMutex: THandle;
begin
  HMutex := CreateMutex (nil, False, 'OneCopyMutex');
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
    end;
  end.
```

If you run this example twice, you'll see that it creates a new, temporary copy of the application (the icon appears in the Taskbar) and then destroys it when the time-out elapses. This approach is certainly more robust than the previous one, but it lacks a feature: how do we enable the existing instance of the application? We still need to find its form, but we can use a better approach.

Searching the Window List

When you want to search for a specific main window in the system, you can use the `EnumWindows` API functions. Enumeration functions are quite peculiar in Windows, because they usually require another function as a parameter. These enumeration functions require a pointer to a function (often described as a *callback* function) as parameter. The idea is that this function is applied to each element of the list (in this case, the list of main windows), until the list ends or the function returns False. Here is the enumeration function from the OneCopy example:

```
function EnumWndProc (hwnd: THandle;
  Param: Cardinal): Bool; stdcall;
var
  ClassName, WinModuleName: string;
  WinInstance: THandle;
begin
  Result := True;
  SetLength (ClassName, 100);
  GetClassName (hwnd, PChar (ClassName), Length (ClassName));
  ClassName := PChar (ClassName);
  if ClassName = TForm1.ClassName then
  begin
    // get the module name of the target window
    SetLength (WinModuleName, 200);
    WinInstance := GetWindowLong (hwnd, GWL_HINSTANCE);
    GetModuleFileName (WinInstance,
      PChar (WinModuleName), Length (WinModuleName));
    WinModuleName := PChar(WinModuleName); // adjust length
    // compare module names
    if WinModuleName = ModuleName then
    begin
      FoundWnd := Hwnd;
      Result := False; // stop enumeration
    end;
  end;
end;
```

This function, called for each nonchild window of the system, checks the name of each window's class, looking for the name of the TForm1 class. When it finds a window with this string in its class name, it uses GetModuleFilename to extract the name of the executable file of the application that owns the matching form. If the module name matches that of the current program (which was extracted previously with similar code), you can be quite sure that you have found a previous instance of the same program. Here is how you can call the enumerated function:

```
var
  FoundWnd: THandle;
  ModuleName: string;
begin
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
    ...
  else
  begin
    // get the current module name
    SetLength (ModuleName, 200);
    GetModuleFileName (HInstance, PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar (ModuleName); // adjust length
    // find window of previous instance
    EnumWindows (@EnumWndProc, 0);
```

Handling User-Defined Window Messages

I've mentioned earlier that the SetForegroundWindow call doesn't work if the main form of the program has been minimized. Now we can solve this problem. You can ask the form of another application—the previous instance of the same program in this case—to restore its main form by sending it a user-defined window message. You can then test whether the form is minimized and post a new user-defined message to the old window. Here is the code; in the OneCopy program, it follows the last fragment shown in the preceding section:

```
    if FoundWnd <> 0 then
    begin
      // show the window, eventually
      if not IsWindowVisible (FoundWnd) then
        PostMessage (FoundWnd, wm_User, 0, 0);
      SetForegroundWindow (FoundWnd);
    end;
```

Again, the PostMessage API function sends a message to the message queue of the application that owns the destination window. In the code of the form, you can add a special function to handle this message:

```
public
  procedure WMUser (var msg: TMessage);
    message wm_User;
```

Now you can write the code of this method, which is simple:

```
procedure TForm1.WMUser (var msg: TMessage);
begin
  Application.Restore;
end;
```

Creating MDI Applications

A common approach for the structure of an application is MDI (Multiple Document Interface). An MDI application is made up of several forms that appear inside a single main form. If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application. But with your favorite word processor, you can probably open several different documents, each in its own child window, because they are MDI applications. All these document windows are usually held by a *frame*, or *application*, window.

NOTE Microsoft is departing more and more from the MDI model stressed in Windows 3 days. Starting with Resource Explorer in Windows 95 and even more with Office 2000, Microsoft tends to use a specific main window for every document, the classic SDI (Single Document Interface) approach. In any case, MDI isn't dead and can sometimes be a useful structure.

MDI in Windows: A Technical Overview

The MDI structure gives programmers several benefits automatically. For example, Windows handles a list of the child windows in one of the pull-down menus of an MDI application, and there are specific Delphi methods that activate the corresponding MDI functionality, to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows:

- The main window of the application acts as a frame or a container.
- A special window, known as the *MDI client*, covers the whole client area of the frame window. This MDI client is one of the Windows predefined controls, just like an edit box or a list box. The MDI client window lacks any specific user-interface element, but it is visible. In fact, you can change the standard system color of the MDI work area (called the Application Background) in the Appearance page of the Display Properties dialog box in Windows.
- There are multiple child windows, of the same kind or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the frame window.

Frame and Child Windows in Delphi

Delphi makes the development of MDI applications easy, even without using the MDI Application template available in Delphi (see the Applications page of the File ➢ New dialog box). You only need to build at least two forms, one with the FormStyle property set to fsMDIForm and the other with the same property set to fsMDIChild. Set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at startup, and you need to provide a way to create one or more child windows. This can be done by adding a menu with a New menu item and writing the following code:

```
var
  ChildForm: TChildForm;
begin
  ChildForm := TChildForm.Create (Application);
  ChildForm.Show;
```

Another important feature is to add a "Window" pull-down menu and use it as the value of the WindowMenu property of the form. This pull-down menu will automatically list all the available child windows. Of course, you can choose any other name for the pull-down menu, but Window is the standard.

To make this program work properly, we can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  WindowMenu := Window1;
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption := ChildForm.Caption + ' ' + IntToStr (Counter);
  ChildForm.Show;
end;
```

You can also open child windows, minimize or maximize each of them, close them, and use the Window pull-down menu to navigate among them. Now suppose that we want to close some of these child windows, to unclutter the client area of our program. Click the Close boxes of some of the child windows and they are minimized! What is happening here? Remember that when you close a window, you generally hide it from view. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, hiding them won't work, because the MDI Window menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi minimizes the MDI child windows when you try to close them. To solve this problem, we need to delete the child windows when they are closed, setting the Action reference parameter of the OnClose event to caFree.

Building a Complete Window Menu

Our first task is to define a better menu structure for the example. Typically the Window pull-down menu has at least three items, titled Cascade, Tile, and Arrange Icons. To handle the menu commands, we can use some of the predefined methods of TForm that can be used only for MDI frames:

- The Cascade method cascades the open MDI child windows. The windows overlap each other. Iconized child windows are also arranged (see ArrangeIcons below).
- The Tile method tiles the open MDI child windows; the child forms are arranged so that they do not overlap. The default behavior is horizontal tiling, although if you have several child windows, they will be arranged in several columns. This default can be changed by using the TileMode property (either tbHorizontal or tbVertical).
- The ArrangeIcons procedure arranges all the iconized child windows. Open forms are not moved.

As a better alternative to calling these methods, you can place an ActionList in the form and add to it a series of predefined MDI actions. The related classes are TWindowArrange, TWindowCascade, TWindowClose, TWindowTileHorizontal, TWindowTileVertical, and TWindowMinimizeAll. The connected menu items will perform the corresponding actions and will be disabled if no child window is available. The MdiDemo example, which we'll look at next, demonstrates the use of the MDI actions, among other things.

There are also some other interesting methods and properties related strictly to MDI in Delphi:

- ActiveMDIChild is a run-time and read-only property of the MDI frame form, and it holds the active child window. The user can change this value by selecting a new child window, or the program can change it using the Next and Previous procedures, which activate the child window following or preceding the currently active one.
- The ClientHandle property holds the Windows handle of the MDI client window, which covers the client area of the main form.
- The MDIChildren property is an array of child windows. You can use this and the MDIChildCount property to cycle among all of the child windows. This can be useful for finding a particular child window or to operate on each of them.

NOTE Note that the internal order of the child windows is the reverse order of activation. This means that the last child window selected is the active window (the first in the internal list), the second-to-last child window selected is the second, and the first child window selected is the last. This order determines how the windows are arranged on the screen. The first window in the list is the one above all others, while the last window is below all others, and probably hidden away. You can imagine an axis (the *z* axis) coming out of the screen toward you. The active window has a higher value for the *z* coordinate and, thus, covers other windows. For this reason, the Windows ordering schema is known as the *z-order*.

The MdiDemo Example

I've built a first example to demonstrate most of the features of a simple MDI application. MdiDemo is actually a full-blown MDI text editor, because each child window hosts a Memo component and can open and save text files. The child form has a `Modified` property used to indicate whether the text of the memo has changed (it is set to True in the handler of the memo's `OnChange` event). `Modified` is set to False in the `Save` and `Load` custom methods and checked when the form is closed (prompting to save the file).

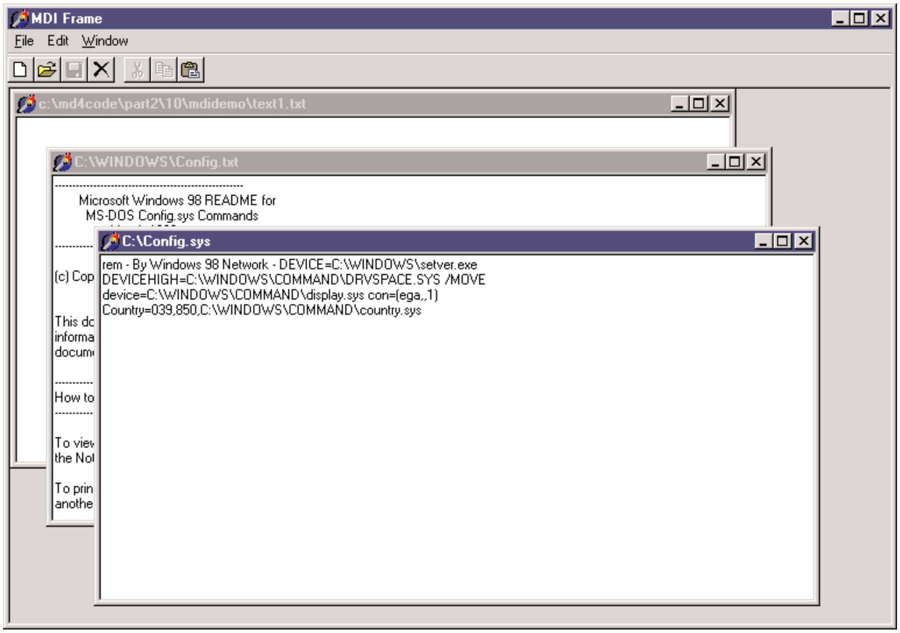
As I've already mentioned, the main form of this example is based on an ActionList component. The actions are available through some menu items and a toolbar, as you can see in Figure 10.4. You can see the details of the ActionList in the source code of the example. Next, I want to focus on the code of the custom actions. Once more, this example demonstrates that using actions makes it very simple to modify the user interface of the program, without writing any extra code. In fact, there is no code directly tied to the user interface.

One of the simplest actions is the `ActionFont` object, which has both an `OnExecute` handler, which uses a FontDialog component, and an `OnUpdate` handler, which disables the action (and hence the associated menu item and toolbar button) when there are no child forms:

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
  if FontDialog1.Execute then
    (ActiveMDIChild as TChildForm).Memo1.Font := FontDialog1.Font;
end;

procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
  ActionFont.Enabled := MDIChildCount > 0;
end;
```

FIGURE 10.4:

The MdiDemo program uses a series of predefined Delphi actions connected to a menu and a toolbar.



```
--------------------------------------------------------
      Microsoft Windows 98 README for
       MS-DOS Config.sys Commands
```

```
-----------
(c) Cop

This dc
informa
docum

-----------
How to
-----------

To viev
the Not

To prin
anothe
```

```
rem - By Windows 98 Network - DEVICE=C:\WINDOWS\setver.exe
DEVICEHIGH=C:\WINDOWS\COMMAND\DRVSPACE.SYS /MOVE
device=C:\WINDOWS\COMMAND\display.sys con=(ega,,1)
Country=039,850,C:\WINDOWS\COMMAND\country.sys
```

The action named New creates the child form and sets a default filename. The Open action calls the `ActionNewExcecute` method prior to loading the file:

```
procedure TMainForm.ActionNewExecute(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption :=
    LowerCase (ExtractFilePath (Application.Exename)) + 'text' +
    IntToStr (Counter) + '.txt';
  ChildForm.Show;
end;

procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    ActionNewExecute (Self);
    (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
  end;
end;
```

The actual file loading is performed by the Load method of the form. Likewise, the Save method of the child form is used by the Save and Save As actions. Notice the OnUpdate handler of the Save action, which enables the action only if the user has changed the text of the memo:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
begin
  // suggest the current file name
  SaveDialog1.FileName := ActiveMDIChild.Caption;
  if SaveDialog1.Execute then
  begin
    // modify the file name and save
    ActiveMDIChild.Caption := SaveDialog1.FileName;
    (ActiveMDIChild as TChildForm).Save;
  end;
end;

procedure TMainForm.ActionSaveUpdate(Sender: TObject);
begin
  ActionSave.Enabled := (MDIChildCount > 0) and
    (ActiveMDIChild as TChildForm).Modified;
end;

procedure TMainForm.ActionSaveExecute(Sender: TObject);
begin
  (ActiveMDIChild as TChildForm).Save;
end;
```

MDI Applications with Different Child Windows

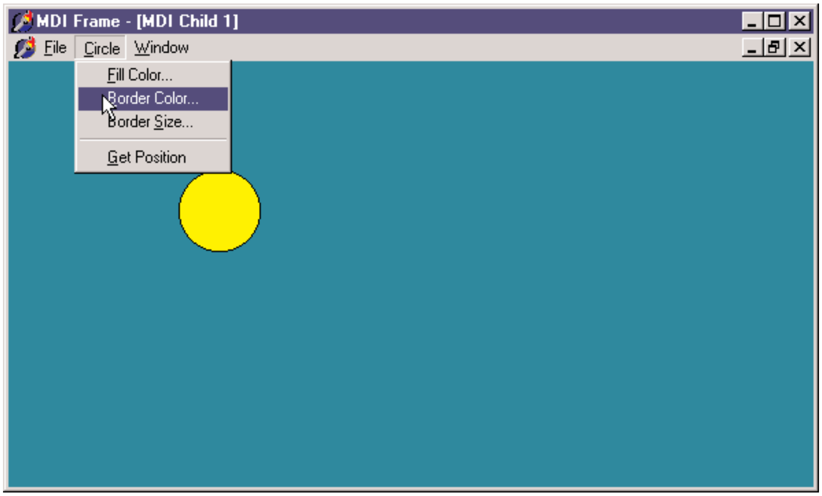
A common approach in complex MDI applications is to include child windows of different kinds (that is, based on different child forms). I will build a new example, called MdiMulti, to highlight some problems you may encounter with this approach. This example has two different types of child forms. The first type will host a circle drawn in the position of the last mouse click, while the second will contain a bouncing square. Another feature I'll add to the main form is a custom background obtained by painting a tiled image in it.

Child Forms and Merging Menus

The first type of child form can display a circle in the position where the user clicked one of the mouse buttons. Figure 10.5 shows an example of the output of the MdiMulti program. The program includes a Circle menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. What is interesting here is that to program the child form, we do not need to consider the existence of other forms or of the frame window. We simply write the code of the form, and that's all. The only special care required is for the menus of the two forms.

FIGURE 10.5:

The output of the MdiMulti example, with a child window that displays circles



If we prepare a main menu for the child form, it will replace the main menu of the frame window when the child form is activated. An MDI child window, in fact, cannot have a menu of its own. But the fact that a child window can't have any menus should not bother you, because this is the standard behavior of MDI applications. You can use the menu bar of the frame window to display the menus of the child window. Even better, you can merge the menu bar of the frame window and that of the child form. For example, in this program, the menu of the child form can be placed between the frame window's File and Window pull-down menus. You can accomplish this using the following `GroupIndex` values:

- File pull-down menu, main form: 1
- Window pull-down menu, main form: 3
- Circle pull-down menu, child form: 2

Using these settings for the menu group indexes, the menu bar of the frame window will have either two or three pull-down menus. At startup, the menu bar has two menus. As soon as you create a child window, there are three menus, and when the last child window is closed (destroyed), the Circle pull-down menu disappears. You should also spend some time testing this behavior by running the program.

The second type of child form shows a moving image. The square, a Shape component, moves around the client area of the form at fixed time intervals, using a Timer component, and bounces on the edges of the form, changing its direction. This turning process is determined by a fairly complex algorithm, which we don't have space to examine. The main point of the example, instead, is to show you how menu merging behaves when you have an MDI frame with child forms of different types. (You can study the source code on the companion CD to see how it works.)

The Main Form

Now we need to integrate the two child forms into an MDI application. The File pull-down menu here has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. The Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the original menu bar of the main form is reset. By using the proper menu group indexes, we let Delphi accomplish everything automatically, as you can see in Figure 10.6.

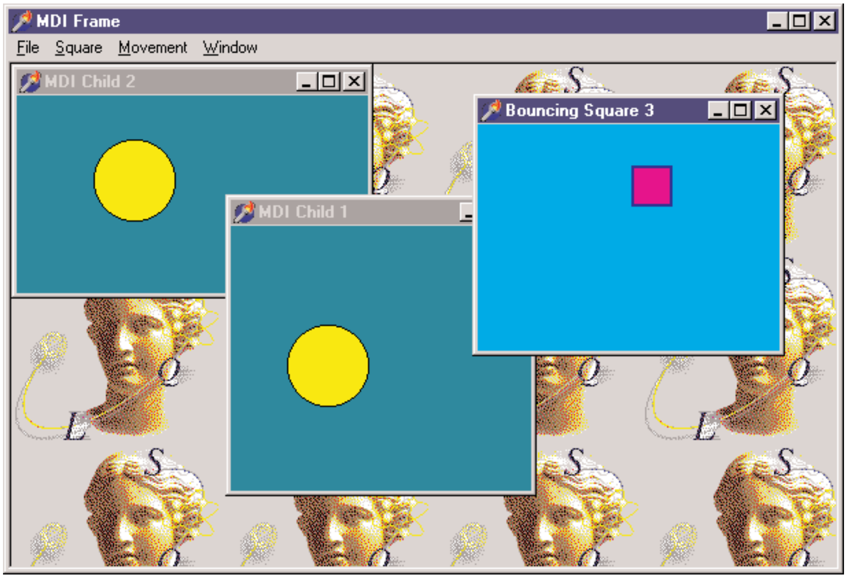


FIGURE 10.6:

The menu bar of the MdiMulti Demo4 application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of Figure 10.5.

I've added a few other menu items in the main form, to close every child window and show some statistics about them. The method related to the Count command scans the MDIChildren array property to count the number of child windows of each kind (using the RTTI operator is):

```
for I := 0 to MDIChildCount - 1 do
  if MDIChildren is TBounceChildForm then
    Inc (NBounce)
  else
    Inc (NCircle);
```

Subclassing the MdiClient Window

Finally, the program includes support for a background-tiled image. The bitmap is taken from an Image component and should be painted on the form in the wm_EraseBkgnd Windows message's handler. The problem is that we cannot simply connect the code to the main form, as a separate window, the MdiClient, covers its surface.

We have no corresponding Delphi form for this window, so how can we handle its messages? We have to resort to a low-level Windows programming technique known as *subclassing*. (In spite of the name, this has little to do with OOP inheritance.) The basic idea is that we can replace the window procedure, which receives all the messages of the window, with a new one we provide. This can be done by calling the SetWindowLong API function and providing the memory address of the procedure, the function pointer.

NOTE A window procedure is a function receiving all the messages for a window. Every window must have a window procedure and can have only one. Even Delphi forms have a window procedure; although this is hidden in the system, it calls the WndProc virtual function, which you can use. But the VCL has a predefined handling of the messages, which are then forwarded to the message-handling methods of a form after some preprocessing. With all this support, you need to handle window procedures explicitly only when working with non-Delphi windows, as in this case.

Unless we have some reason to change the default behavior of this system window, we can simply store the original procedure and call it to obtain a default processing. The two function pointers referring to the two procedures (the old and the new one) are stored in two local fields of the form:

```
private
  OldWinProc, NewWinProc: Pointer;
  procedure NewWinProcedure (var Msg: TMessage);
```

The form also has a method we'll use as a new window procedure, with the actual code used to paint on the background of the window. Because this is a method and not a plain window procedure, the program has to call the MakeObjectInstance method to add a prefix to the method and let the system use it as if it were a function. All this description is summarized by just two complex statements:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  NewWinProc := MakeObjectInstance (NewWinProcedure);
  OldWinProc := Pointer (SetWindowLong (ClientHandle, gwl_WndProc, Cardinal
    (NewWinProc)));
  OutCanvas := TCanvas.Create;
end;
```

The window procedure we install calls the default one. Then, if the message is wm_EraseBkgnd and the image is not empty, we draw it on the screen many times using the Draw method of a temporary canvas. This canvas object is created when the program starts (see the code above) and connected to the handle passed as wParam parameter by the message. With this approach, we don't have to create a new TCanvas object for every background painting operation requested, thus saving a little time in the frequent operation. Here is the code, which produces the output already seen in Figure 10.6:

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
  BmpWidth, BmpHeight: Integer;
  I, J: Integer;
begin
  // default processing first
  Msg.Result := CallWindowProc (OldWinProc, ClientHandle, Msg.Msg, Msg.wParam,
    Msg.lParam);

  // handle background repaint
  if Msg.Msg = wm_EraseBkgnd then
  begin
    BmpWidth := MainForm.Image1.Width;
    BmpHeight := MainForm.Image1.Height;
    if (BmpWidth <> 0) and (BmpHeight <> 0) then
    begin
      OutCanvas.Handle := Msg.wParam;
      for I := 0 to MainForm.ClientWidth div BmpWidth do
        for J := 0 to MainForm.ClientHeight div BmpHeight do
          OutCanvas.Draw (I * BmpWidth, J * BmpHeight,
            MainForm.Image1.Picture.Graphic);
    end;
  end;
end;
```

Visual Form Inheritance

When you need to build two or more similar forms, possibly with different event handlers, you can use dynamic techniques, hide or create new components at run time, change event handlers, and use `if` or `case` statements. Or you can apply the object-oriented techniques, thanks to visual form inheritance. In short, instead of creating a form based on `TForm`, you can inherit a form from an existing one, adding new components or altering the properties of the existing ones. But what is the real advantage of visual form inheritance?

Well, this mostly depends on the kind of application you are building. If it has multiple forms, some of which are very similar to each other or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the forms of an application.

You can also use visual form inheritance to customize an application for different clients, without duplicating any source code or form definition code; just inherit the specific versions for a client from the standard forms. Remember that the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method in a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

NOTE Delphi includes another feature, frames, which resembles visual form inheritance. In both cases, you can work at design time on two versions of a form/frame. However, in visual form inheritance, you are defining two different classes (parent and derived), whereas with frames, you work on a class and an instance. Frames will be discussed in detail later in this chapter.

Inheriting from a Base Form

The rules governing visual form inheritance are quite simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

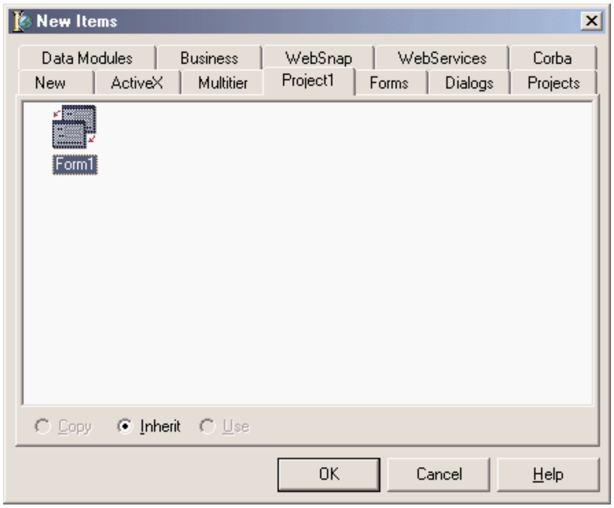
Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command of the Object Inspector. The same thing is accomplished by setting the two properties to the same value and recompiling the code. After modifying multiple properties, you can resynchronize them all to the base version by applying the Revert to Inherited command of the component's local menu.

Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To describe how visual form inheritance works, I've built a very simple example, called VFI. I'll describe step-by-step how to build it. First, start a new project, and add four buttons to its main form. Then select File ➢ New ➢ Other, and choose the page with the name of the project in the New Items dialog box (see Figure 10.7).

FIGURE 10.7:

The New Items dialog box allows you to create an inherited form.



In the New Items dialog, you can choose the form from which you want to inherit. The new form has the same four buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

And here is its initial class declaration, where you can see that the base class is not the usual TForm but the actual base class form:

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
    public
      { Public declarations }
    end;
```

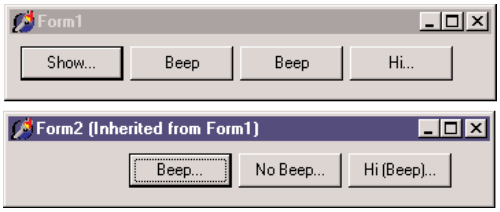
Notice the presence of the inherited keyword in the textual description; also notice that the form indeed has some components, although they are defined in the base class form. If you move the form and add the caption of one of the buttons, the textual description will change accordingly:

```
inherited Form2: TForm2
  Left = 313
  Top = 202
  Caption = 'Form2'
  inherited Button2: TButton
    Caption = 'Beep...'
  end
end
```

Only the properties with a different value are listed (and by removing these properties from the textual description of the inherited form, you can reset them to the value of the base form, as I mentioned before). I've actually changed the captions of most buttons, as you can see in Figure 10.8.

FIGURE 10.8:

The two forms of the VFI example at run time



Each of the buttons of the first form has an OnClick handler, with simple code. The first button shows the inherited form calling its Show method; the second and the third buttons call the Beep procedure; and the last button displays a simple message.

What happens in the inherited form? First we should remove the Show button, because the secondary form is already visible. However, we cannot delete a component from an inherited form. An alternative solution is to leave the component there but set its Visible property to False. The button will still be there but not visible (as you can guess from Figure 10.8). The other three buttons will be visible but with different handlers. This is simple to accomplish. If you select the OnClick event of a button in the inherited form (by double-clicking it), you'll get an empty method slightly different from the default one:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
end;
```

The inherited keyword stands for a call to the corresponding event handler of the base form. This keyword is always added by Delphi, even if the handler is not defined in the parent class (and this is reasonable, because it might be defined later) or if the component is not present in the parent class (which doesn't seem like a great idea to me). It is very simple to execute the code of the base form and perform some other operations:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hi');
end;
```

This is not the only choice. An alternative approach is to write a brand-new event handler and not execute the code of the base class, as I've done for the third button of the VFI example: To accomplish this, simply remove the inherited keyword. Still another choice includes calling a base-class method after some custom code has been executed, calling it when a condition is met, or calling the handler of a different event of the base class, as I've done for the fourth button:

```
procedure TForm2.Button4Click(Sender: TObject);
begin
  inherited Button3Click (Sender);
  inherited;
end;
```

You probably won't do this very often, but you must be aware that you can. Of course, you can consider each method of the base form as a method of your form, and call it freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at real-world examples more complex than this book has room to explore. There is something else I want to show you here: *visual form polymorphism*.

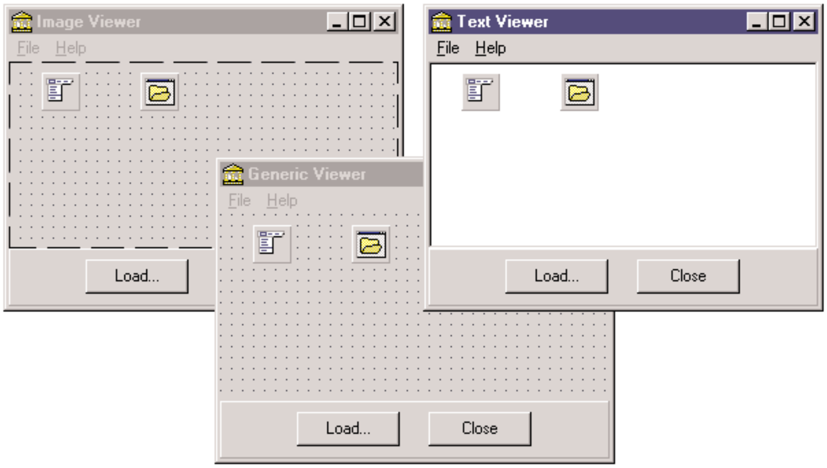
Polymorphic Forms

The problem is simple. If you add an event handler to a form and then change it in an inherited form, there is no way to refer to the two methods using a common variable of the base class, because the event handlers use static binding by default.

Confusing? Here is an example, which is intended for experienced Delphi programmers. Suppose you want to build a bitmap viewer form and a text viewer form in the same program. The two forms have similar elements, a similar toolbar, a similar menu, an OpenDialog component, and different components for viewing the data. So you decide to build a base-class form containing the common elements and inherit the two forms from it. You can see the three forms at design time in Figure 10.9.

FIGURE 10.9:

The base-class form and the two inherited forms of the PoliForm example at design time



The main form contains a toolbar panel with a few buttons (real toolbars apparently have a few problems with visual form inheritance), a menu, and an open dialog component. The two inherited forms have only minor differences, but they feature a new component, either an image viewer (`TImage`) or a text viewer (`TMemo`). They also modify the settings of the OpenDialog component, to refer to different types of files.

The main form includes some common code. The Close button and the File ➢ Close command call the `Close` method of the form. The Help ➢ About command shows a simple message box. The Load button of the base form has the following code:

```
procedure TViewerForm.ButtonLoadClick(Sender: TObject);
begin
  ShowMessage ('Error: File-loading code missing');
end;
```

The File ➢ Load command, instead, calls another method:

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
  LoadFile;
end;
```

This method is defined in the `TViewerForm` class as a virtual abstract method (so that the class of the base form is actually an abstract class). Because this is an abstract method, we will need to redefine it (and override it) in the inherited forms. The code of this `LoadFile` method simply uses the `OpenDialog1` component to ask the user to select an input file and loads it into the image component:

```
procedure TImageViewerForm.LoadFile;
begin
```

```
    if OpenDialog1.Execute then
      Image1.Picture.LoadFromFile (OpenDialog1.Filename);
  end;
```

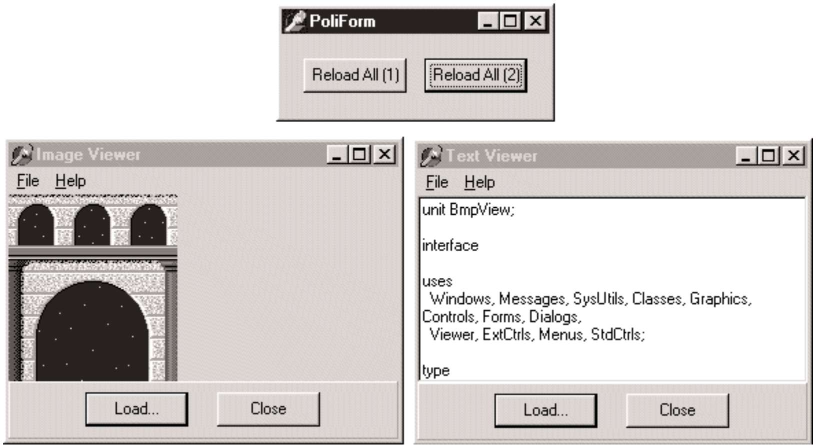
The other inherited class has similar code, loading the text into the memo component. The project has one more form, a main form with two buttons, used to reload the files in each of the viewer forms. The main form is the only form created by the project when it starts. The generic viewer form is never created: it is only a generic base class, containing common code and components of the two subclasses. The forms of the two subclasses are created in the `OnCreate` event handler of the main form:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  FormList [1] := TTextViewerForm.Create (Application);
  FormList [2] := TImageViewerForm.Create (Application);
  for I := 1 to 2 do
    FormList[I].Show;
end;
```

See Figure 10.10 for the resulting forms (with text and image already loaded in the viewers). `FormList` is a *polymorphic* array of generic `TViewerForm` objects, declared in the `TMainForm` class.

FIGURE 10.10:

The PoliForm example at run time



```
unit BmpView;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs,
  Viewer, ExtCtrls, Menus, StdCtrls;

type
```

Note that to make this declaration in the class, you need to add the Viewer unit (but not the specific forms) in the `uses` clause of the interface portion of the main form. The array of forms is used to load a new file in each viewer form when one of the two buttons is pressed. The handlers of the two buttons' OnClick events use different approaches:

```
procedure TMainForm.ReloadButton1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList [I].ButtonLoadClick (Self);
end;

procedure TMainForm.ReloadButton2Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList [I].LoadFile;
end;
```

The second button simply calls a virtual method, and it will work without any problem. The first button calls an event handler and will always reach the generic TFormView class (displaying the error message of its ButtonLoadClick method). This happens because the method is static, not virtual.

Is there a way to make this approach work? Sure. Declare the ButtonLoadClick method of the TFormView class as virtual, and declare it as overridden in each of the inherited form classes, as we do for any other virtual method:

```
type
  TViewerForm = class(TForm)
    // components and plain methods...
    procedure ButtonLoadClick(Sender: TObject); virtual;
  public
    procedure LoadFile; virtual; abstract;
  end;
...
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject); override;
  public
    procedure LoadFile; override;
  end;
```

This trick really works, although it is never mentioned in the Delphi documentation. This ability to use virtual event handlers is what I actually mean by visual form polymorphism. In other (more technical) words, you can assign a virtual method to an event property, which will take the address of the method according to the instance available at run time.

Understanding Frames

Chapter 1 briefly discussed frames, which were introduced in Delphi 5. We've seen that you can create a new frame, place some components in it, write some event handlers for the components, and then add the frame to a form. In other words, a frame is similar to a form, but it defines only a portion of a window, not a complete window. This is certainly not a feature worth a new construct. The totally new element of frames is that you can create multiple instances of a frame at design time, and you can modify the class and the instance at the same time. This makes frames an effective tool for creating customizable composite controls at design time, something close to a visual component-building tool.

In visual form inheritance you can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one, unless this overrides some property or event. With frames, you work on a class (as usual in Delphi), but the difference is that you can also customize one or more instances of the class at design time. When you work on a form, you cannot change a property of the `TForm1` class for the `Form1` object at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, in fact, you can customize each of the instances at design time. Wasn't this already possible with component templates? It was, but component templates were based on the concept of copying and pasting some components and their code. There was no way to change the original definition of the template and see the effect in every place it was used. That is what happens with frames (and in a different way with visual form inheritance); changes to the original version (the class) are reflected in the copies (the instances).

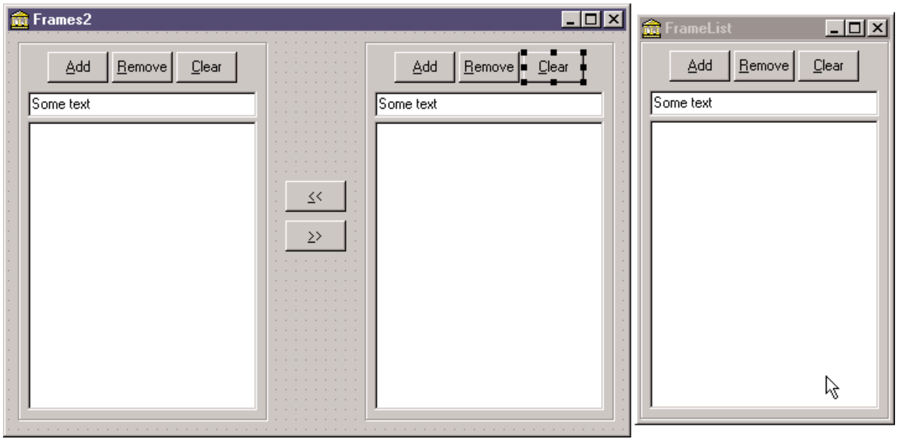
Let's discuss a few more elements of frames with an example from the CD, called Frames2. This program has a frame with a list box, an edit box, and three buttons with simple code operating on the components. The frame also has a bevel aligned to its client area, because frames have no border. Of course, the frame has also a corresponding class, which looks like a form class:

```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

What is different is that you can add the frame to a form. I've used two instances of the frame in the example (as you can see in Figure 10.11) and modified the behavior slightly. The first instance of the frame has the list box items sorted. When you change a property of a component of a frame, the DFM file of the hosting form will list the differences, as it does with visual form inheritance:

FIGURE 10.11:

A frame and two instances of it at design time, in the Frames2 example



```
object FormFrames: TFormFrames
  Caption = 'Frames2'
  inline FrameList1: TFrameList
    Left = 8
    Top = 8
    inherited ListBox: TListBox
      Sorted = True
    end
  end
  inline FrameList2: TFrameList
    Left = 232
    Top = 8
    inherited btnClear: TButton
      OnClick = FrameList2btnClearClick
    end
  end
end
```

As you can see from the listing, the DFM file for a form that has frames uses a new DFM keyword, `inline`. The references to the modified components of the frame, instead, use the `inherited` keyword, although this term is used with an extended meaning. `inherited` here doesn't refer to a base class we are inheriting from, but to the class we are instancing (or inheriting) an object from. It was probably a good idea, though, to use an existing feature of visual form inheritance and apply it to the new context. The effect of this approach, in fact, is that you can use the Revert to Inherited command of the Object Inspector or of the form to cancel the changes and get back to the default value of properties.

Notice also that unmodified components of the frame class are not listed in the DFM file of the form using the frame, and that the form has two frames with different names, but the components on the two frames have the same name. In fact, these components are not owned by the form, but are owned by the frame. This implies that the form has to reference those components through the frame, as you can see in the code for the buttons that copy items from one list box to the other:

```
procedure TFormFrames.btnLeftClick(Sender: TObject);
begin
  FrameList1.ListBox.Items.AddStrings (FrameList2.ListBox.Items);
end;
```

Finally, besides modifying properties of any instance of a frame, you can change the code of any of its event handlers. If you double-click one of the buttons of a frame while working on the form (not on the stand-alone frame), Delphi will generate this code for you:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  FrameList2.btnClearClick(Sender);
end;
```

The line of code automatically added by Delphi corresponds to a call to the inherited event handler of the base class in visual form inheritance. This time, however, to get the default behavior of the frame we need to call an event handler and apply it to a specific instance—the frame object itself. The current form, in fact, doesn't include this event handler and knows nothing about it.

Whether you leave this call in place or remove it depends on the effect you are looking for. In the example I've decided to conditionally execute the default code, depending on the user confirmation:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  if MessageDlg ('OK to empty the list box?', mtConfirmation,
      [mbYes, mbNo], 0) = idYes then
    // execute standard frame code
    FrameList2.btnClearClick(Sender);
end;
```

TIP By the way, note that because the event handler has some code, leaving it empty and saving the form won't remove it as usual: in fact, it isn't empty! Instead, if you simply want to omit the default code for an event, you need to add at least a comment to it, to avoid it being automatically removed by the system!

Frames and Pages

When you have a dialog box with many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared in a single form. Also, creating all these components (and initializing them) might result in a delay in the display of the dialog box. Frames actually don't reduce the construction and initialization time of equivalently loaded forms; quite the contrary, as loading frames is more complicated for the streaming system than loading simple components. However, using frames you can load only the visible pages of a multipage dialog box, reducing the *initial* load time, which is what the user perceives.

Frames can solve both of these issues. First, you can easily divide the code of a single complex form into one frame per page. The form will simply host all of the frames in a PageControl. This certainly helps you to have simpler and more focused units and makes it simpler to reuse a specific page in a different dialog box or application. Reusing a single page of a PageControl without using a frame or an embedded form, in fact, is far from simple.

As an example of this approach I've built the FramePag example, which has some frames placed inside the three pages of a PageControl, as you can see in Figure 10.12. All of the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them. Actually two of the pages have the same frame, but the two instances of the frame have some differences at design time. The frame, called `Frame3` in the example, has a list box that is populated with a text file at startup, and has buttons to modify the items in the list and saves them to a file. The filename is placed inside a label, so that you can easily select a file for the frame at design time by changing the `Caption` of the label.

TIP Being able to use multiple instances of a frame is one of the reasons this technique was introduced, and customizing the frame at design time is even more important. Because adding properties to a frame and making them available at design time requires some customized and complex code, it is nice to use a component to host these custom values. You have the option of hiding these components (such as the label in our example) if they don't pertain to the user interface.

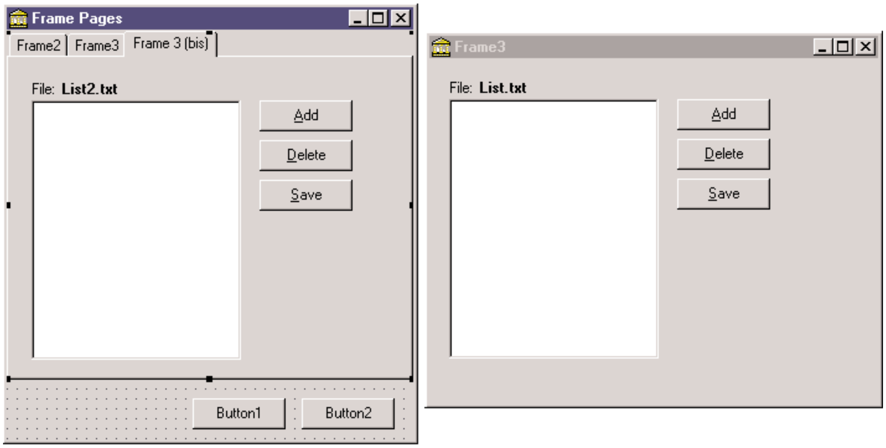


FIGURE 10.12: Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks.

In the example, we need to load the file when the frame instance is created. Because frames have no `OnCreate` event, our best choice is probably to override the `CreateWnd` method. Writing a custom constructor, in fact, doesn't work as it is executed too early—before the specific label text is available. Here is the frame class code:

```
type
  TFrame3 = class(TFrame)
    ...
  public
    procedure CreateWnd; override;
```

Within the `CreateWnd` method, we simply load the list box content from a file.

Multiple Frames with No Pages

Another approach is to avoid creating all of the pages along with the form hosting them. This can be accomplished by leaving the PageControl empty and creating the frames only when a page is displayed. Actually, when you have frames on multiple pages of a PageControl, the windows for the frames are created only when they are first displayed, as you can find out by placing a breakpoint in the creation code of the last example.

As an even more radical approach, you can get rid of the page controls and use a TabControl. Used this way, the tab has no connected tab sheets (or pages) but can display only one set of information at a time. For this reason, we'll need to create the current frame and destroy the previous one or simply hide it by setting its `Visible` property to False or by calling the `BringToFront` of the new frame. Although this sounds like a lot of work, in a large application this technique can be worth it for the reduced resource and memory usage you can obtain by applying it.

To demonstrate this approach, I've built an example similar to the previous one, this time based on a TabControl and dynamically created frames. The main form, visible at run time in Figure 10.13, has only a TabControl with one page for each frame:

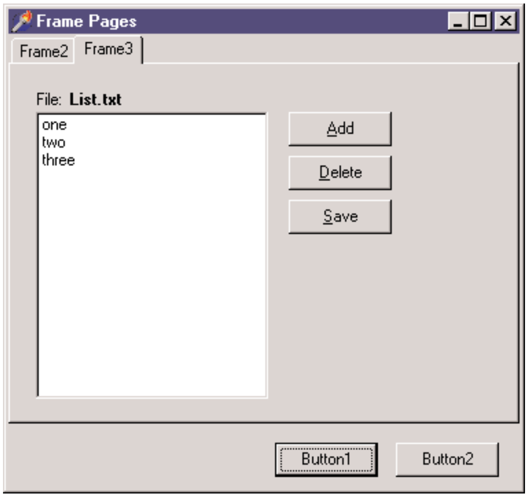


FIGURE 10.13: The first page of the FrameTab example at run time. The frame inside the tab is created at run time.

```
object Form1: TForm1
  Caption = 'Frame Pages'
  OnCreate = FormCreate
  object Button1: TButton...
  object Button2: TButton...
  object Tab: TTabControl
    Anchors = [akLeft, akTop, akRight, akBottom]
    Tabs.Strings = ( 'Frame2' 'Frame3' )
    OnChange = TabChange
  end
end
```

I've given each tab a caption corresponding to the name of the frame, because I'm going to use this information to create the new pages. When the form is created, and whenever the user changes the active tab, the program gets the current caption of the tab and passes it to the custom ShowFrame method. The code of this method, listed below, checks whether the requested frame already exists (frame names in this example follow the Delphi standard of having a number appended to the class name), and then brings it to the front. If the frame doesn't exist, it uses the frame name to find the related frame class, creates an object of that class, and assigns a few properties to it. The code makes extensive use of class references and dynamic creation techniques:

```
type
  TFrameClass = class of TFrame;

procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + '1') as TFrame;
  if not Assigned (Frame) then
  begin
    FrameClass := TFrameClass (FindClass ('T' + FrameName));
    Frame := FrameClass.Create (Self);
    Frame.Parent := Tab;
    Frame.Visible := True;
    Frame.Name := FrameName + '1';
  end;
  Frame.BringToFront;
end;
```

To make this code work, you have to remember to add a call to `RegisterClass` in the initialization section of each unit defining a frame.

Base Forms and Interfaces

We have seen that when you need two similar forms inside an application, you can use visual form inheritance to inherit one from the other or both of them from a common ancestor. The advantage of visual form inheritance is that you can use it to inherit the visual definition, the DFM. However, this is not always requested.

At times, you might want several forms to exhibit a common behavior, or respond to the same commands, without having any shared component or user interface elements. Using visual form inheritance with a base form that has no extra components makes little sense to me. I rather prefer defining my own custom form class, inherited from `TForm`, and then manually editing the form class declarations to inherit from this custom base form class instead of the standard one. If all you need is to define some shared methods, or override `TForm` virtual methods in a consistent way, defining custom form classes can be a very good idea.

Using a Base Form Class

A simple demonstration of this technique is available in the FormIntf demo, showcasing also the use of interfaces for forms. In a new unit, called SaveStatusForm, I've defined the following form class (with no related DFM file—don't use the New Form command, but create a new unit and type the code in it):

```
type
  TSaveStatusForm = class (TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

The two overridden methods are called at the same time of the event handler, so that I can attach extra code (allowing the event handler to be defined as usual). Inside the two methods I simply load or save the form position inside an INI file of the application, in a section marked with the form caption. Here is the code of the two methods:

```
procedure TSaveStatusForm.DoCreate;
var
  Ini: TIniFile;
begin
  inherited;
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Left := Ini.ReadInteger(Caption, 'Left', Left);
  Top := Ini.ReadInteger(Caption, 'Top', Top);
  Width := Ini.ReadInteger(Caption, 'Width', Width);
  Height := Ini.ReadInteger(Caption, 'Height', Height);
  Ini.Free;
end;

procedure TSaveStatusForm.DoDestroy;
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Ini.WriteInteger(Caption, 'Left', Left);
  Ini.WriteInteger(Caption, 'Top', Top);
  Ini.WriteInteger(Caption, 'Width', Width);
  Ini.WriteInteger(Caption, 'Height', Height);
  Ini.Free;
  inherited;
end;
```

Again, this is a simple common behavior for your forms, but you can define a very complex class here. To use this as a base class of the forms you create, simply let Delphi create the forms as usual (with no inheritance) and then update the form declaration to something like:

```
type
  TFormBitmap = class(TSaveStatusForm)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    ...
```

Simple as it seems, this is a very powerful technique, as all you need to do is change the definition of the forms of your application to refer to this base class. If even this is too tedious, as you might want to change this base class in the life of your program, you can use an extra trick, "interposer" classes.

INI Files and the Registry in Delphi

If you want to save information about the status of an application in order to restore it the next time the program is executed, you can use the explicit support that Windows provides for storing this kind of information. INI files, the old Windows standard, are once again the preferred way to save application data. The alternative is the Registry, which is still quite popular. Delphi provides ready-to-use classes to manipulate both.

The *TIniFile* Class

For INI files, Delphi has a `TIniFile` class. Once you have created an object of this class and connected it to a file, you can read and write information to it. To create the object, you need to call the constructor, passing a filename to it, as in the following code:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create ('myprogram.ini');
```

There are two choices for the location of the INI file. The code just listed will store the file in the Windows directory or a user folder for settings in Windows 2000. To store data locally to the application (as opposed to local to the current user), you should provide a full path to the constructor.

INI files are divided into sections, each indicated by a name enclosed in square brackets. Each section can contain multiple items of three possible kinds: strings, integers, or Booleans. The `TIniFile` class has three Read methods, one for each kind of data: `ReadBool`, `ReadInteger`, and `ReadString`. There are also three corresponding methods to write the data: `WriteBool`, `WriteInteger`, and `WriteString`. Other methods allow you to read or erase a whole section. In the Read methods, you can also specify a default value to be used if the corresponding entry doesn't exist in the INI file.

Continued on next page

By the way, notice that Delphi uses INI files quite often, but they are disguised with different names. For example, the desktop (`.dsk`) and options (`.dof`) files are structured as INI files.

The *TRegistry* and *TRegIniFile* classes

The Registry is a hierarchical database of information about the computer, the software configuration, and the user preferences. Windows has a set of API functions to interact with the Registry; you basically open a key (or folder) and then work with subkeys (or subfolders) and with values (or items), but you must be aware of the structure and the details of the Registry.

Delphi provides basically two approaches to the use of the Registry. The `TRegistry` class provides a generic encapsulation of the Registry API, while the `TRegIniFile` class provides the interface of the `TIniFile` class but saves the data in the Registry. This class is the natural choice for portability between INI-based and Registry-based versions of the same program. When you create a `TRegIniFile` object, your data ends up in the current user information, so you'll generally use a constructor like this:

```
IniFile := TRegIniFile.Create ('Software\MyCompany\MyProgram');
```

By using the `TIniFile` and the `TRegistryIniFile` classes offered by the VCL, you can move from one model of local and per-user storage to the other. Not that I think you should use the Registry a lot, as the idea of having a centralized repository for the settings of each application was a architectural error. Even Microsoft acknowledges this (without really admitting the error) by suggesting, in the Windows 2000 Compatibility Requirements, that you not use the Registry anymore for applications settings, but go back to the use of INI files.

An Extra Trick: Interposer Classes

In contrast with Delphi VCL components, which must have unique names, Delphi classes in general must be unique only within their unit. This means you can have two different units defining a class with the same name. This looks really weird, at first sight, but can be useful. For example, Borland is using this technique to provide compatibility between VCL and VisualCLX classes. Both have a `TForm` class, one defined in the Forms unit and the other in the QForms unit. How can this be interesting for the topic discussed here?

NOTE This technique is actually much older than CLX/VCL. For example, the service and control panel applet units define their own `TApplication` object, which is not related to the `TApplication` used by VCL visual GUI applications and defined in the Forms unit.

There is a technique that I've seen mentioned with the name "interposer classes" in an old issue of *The Delphi Magazine*, which suggested replacing standard Delphi class names with your own versions, having the same class name. This way you can use Delphi designer referring to Delphi standard components at design time, but using your own classes at run time.

The idea is simple. In the SaveStatusForm unit, I could have defined the new form class as follows:

```
type
  TForm = class (Forms.TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

This class is called `TForm`, and inherits from `TForm` of the Forms unit (this last reference is compulsory to avoid a kind of recursive definition). In the rest of the program, at this point, you don't need to change the class definition for your form, but simply add the unit defining the interposer class (the SaveStatusForm unit in this case) in the `uses` statement *after* the unit defining the Delphi class. The order of the unit in the `uses` statement is important here, and the reason some people criticize this technique, as it is hard to know what is going on. I have to agree: I find interposer classes handy at times (more for components than for forms, I have to say), but their use makes programs less readable and at times even harder to debug.

Using Interfaces

Another technique, which is slightly more complex but even more powerful than the definition of a common base form class, is to have forms that implement specific interfaces. This way you can have forms implementing one or more of these interfaces, query each form for the interfaces it implements, and call the supported methods.

As an example (available in the same FormIntf program I began discussing in the last section), I've defined a simple interface for loading and storing:

```
type
  IFormOperations = interface
    ['{DACFDB76-0703-4A40-A951-10D140B4A2A0}']
    procedure Load;
    procedure Save;
  end;
```

Each form can optionally implement this interface, as the following `TFormBitmap` class:

```
type
  TFormBitmap = class(TForm, IFormOperations)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    SavePictureDialog1: TSavePictureDialog;
  public
    procedure Load;
    procedure Save;
  end;
```

You can see the actual code of the example for the code of the Load and Save methods, which use the standard dialog boxes to load or save the image. (In the example's code, the form also inherits from the TSaveStatusForm class.)

When an application has one or more forms implementing interfaces, you can apply a given interface method to all the forms supporting it, with code like this (extracted from the main form of the IntfForm example):

```
procedure TFormMain.btnLoadClick(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to Screen.FormCount - 1 do
    if Supports (Screen.Forms [i], IFormOperations) then
      (Screen.Forms [i] as IFormOperations).Load;
end;
```

Consider a business application when you can synchronize all of the forms to the data of a specific company, or a specific business event. And consider also that, unlike inheritance, you can have several forms each implementing multiple interfaces, with unlimited combinations. This is why using an architecture like this can improve a complex Delphi application a great deal, making it much more flexible and easier to adapt to implementation changes.

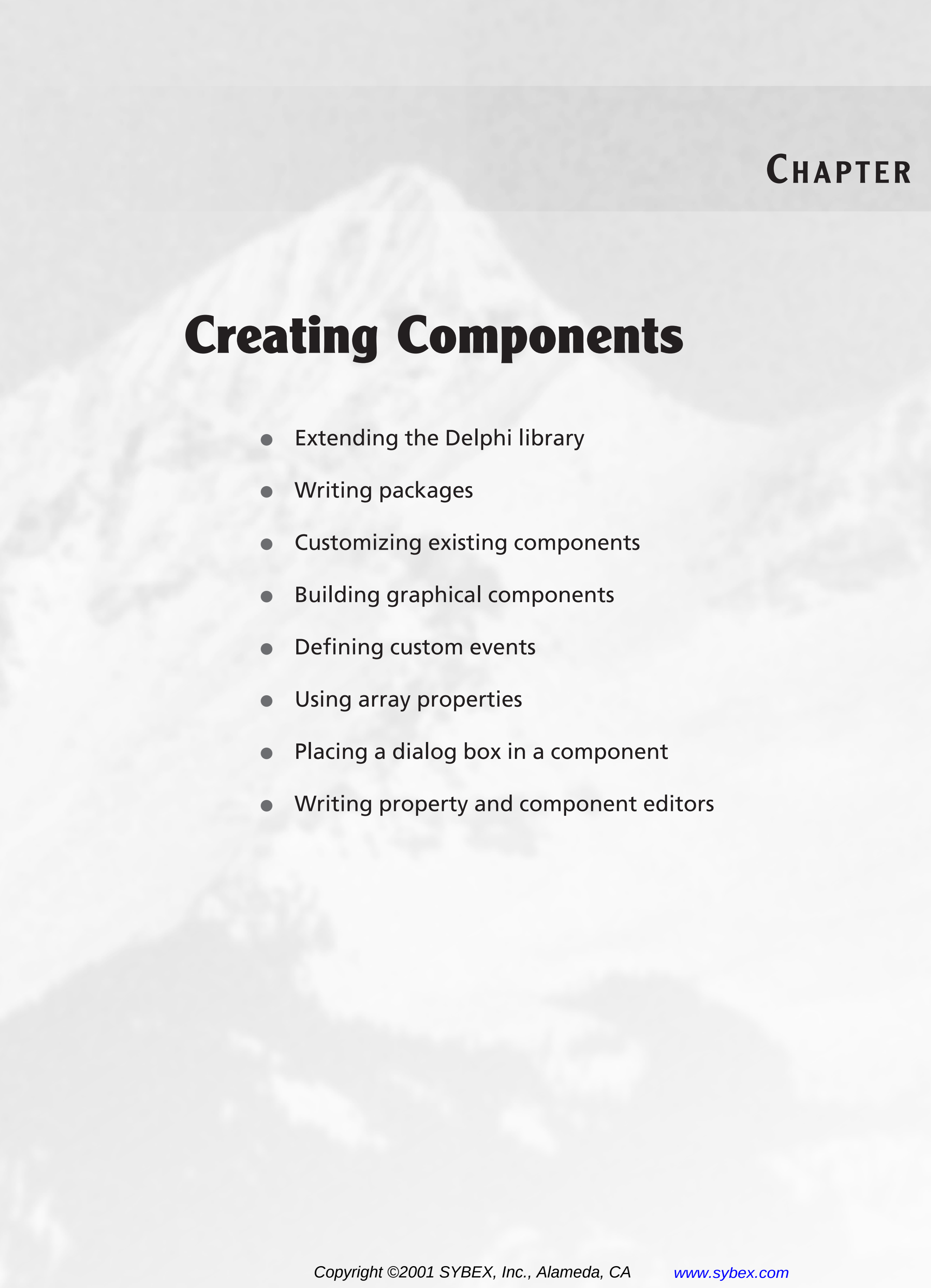
What's Next?

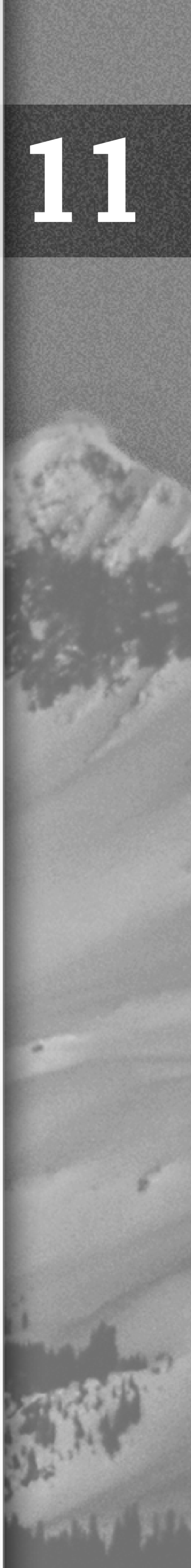
After the detailed description of forms and secondary forms in the previous chapters, I have focused on the architecture of applications, discussing both how Delphi's Application object works and how we can structure applications with multiple forms.

In particular, I've discussed MDI, visual form inheritance, and frames. Toward the end I also discussed custom architectures, with form inheritance and interfaces. Now we can move forward to another key element of non-trivial Delphi applications: building custom components to use within your programs. It is possible to write a specific book about this, so the description won't be exhaustive, but you should be able to get a comprehensive overview.

Another element related to the architecture of Delphi applications is the use of packages, which I'll introduce as a technology related to components but which really goes beyond this. In fact, you can structure the code of a large application in multiple packages, containing forms and other units. The development of programs based on multiple executable files, libraries, and packages, is discussed in Chapter 12.

After this further step, I will start delving into Delphi database programming, certainly another key element of the Borland development environment.





CHAPTER 11

Creating Components

- Extending the Delphi library
- Writing packages
- Customizing existing components
- Building graphical components
- Defining custom events
- Using array properties
- Placing a dialog box in a component
- Writing property and component editors

While most Delphi programmers are probably familiar with using existing components, at times it can also be useful to write our own components or to customize existing ones. One of the most interesting aspects of Delphi is that creating components is simple. For this reason, even though this book is intended for Delphi application programmers and not Delphi tool writers, this chapter will cover the topic of creating components and introduce Delphi add-ins, such as component and property editors.

This chapter gives an overview of writing Delphi components and presents some simple examples. There is not enough space to present very complex components, but the ideas in this chapter will cover all the basics to get you started.

NOTE You'll find a more information about writing components in Chapter 18, "Writing Database Components," including how to build data-aware components.

Extending the Delphi Library

Delphi components are classes, and the Visual Components Library (VCL) is the collection of all the classes defining Delphi components. Each time you add a new package with some components to Delphi, you actually extend VCL with a new class. This new class will be derived from one of the existing component-related classes or the generic `TComponent` class, adding new capabilities to those it inherits.

You can derive a new component from an existing component or from an *abstract component class*—one that does not correspond to a usable component. The VCL hierarchy includes many of these intermediate classes (often indicated with the `TCustom` prefix in their name) to let you choose a default behavior for your new component and to change its properties.

Component Packages

Components are added to component packages. Each component package is basically a DLL (a dynamic link library) with a BPL extension (which stands for Borland Package Library).

Packages come in two flavors: design-time packages used by the Delphi IDE and run-time packages optionally used by applications. The design-only or run-only package option determines the package's type. When you attempt to install a package, the IDE checks whether it has the design-only or run-only flags, and decides whether to let the user install the package and whether it should be added to the list of run-time packages. Since there are two nonexclusive options, each with two possible states, there are four different kinds of component packages—two main variations and two special cases:

- Design-only component packages can be installed in the Delphi environment. These packages usually contain the design-time parts of a component, such as its property editors and the registration code. Often they can also contain the components themselves, although this is not the most professional approach. The code of the components of a design-only package is usually statically linked into the executable file, using the code of the corresponding Delphi Compiled Unit (DCU) files. Keep in mind, however, that it is also technically possible to use a design-only package as a run-time package.
- Run-only component packages are used by Delphi applications at run time. They cannot be installed in the Delphi environment, but they are automatically added to the list of run-time packages when they are required by a design-only package you install. Run-only packages usually contain the code of the component classes, but no design-time support (this is done to minimize the size of the component libraries you ship along with your executable file). Run-only packages are important because they can be freely distributed along with applications, but other Delphi programmers won't be able to install them in the environment to build new programs.
- Plain component packages (having neither the design-only nor the run-only option set) cannot be installed and will not be added to the list of run-time packages automatically. This might make sense for utility packages used by other packages, but they are certainly rare.
- Packages with both flags set can be installed and are automatically added to the list of run-time packages. Usually these packages contain components requiring little or no design-time support (apart from the limited component registration code). Keep in mind, however, that users of applications built with these packages can use them for their own development.

TIP The filenames of Delphi's own design-only packages start with the letters *DCL* (for example, `DCLSTD60.BPL`); filenames of run-only packages start with the letters VCL (for example, `VCL60.BPL`). You can follow the same approach for your own packages, if you want.

In Chapter 1, "The Delphi 6 IDE," we discussed the effect of packages on the size of a program's executable file. Now we'll focus on building packages, since this is a required step in creating or installing components in Delphi.

When you compile a run-time package, you produce both a dynamic link library with the compiled code (the BPL file) and a file with only symbol information (a DCP file), including no compiled machine code. The latter file is used by the Delphi compiler to gather symbol information about the units that are part of the package without having access to the unit (DCU) files, which contain both the symbol information and the compiled machine code. This reduces compilation time and allows you to distribute just the packages without the precompiled unit files. The precompiled units are still required to statically link the components into an application. Distribution of precompiled DCU files (or source code) may make sense depending on the kind of components you develop. We'll see how to create a package after we've discussed some general guidelines and built our very first component.

NOTE DLLs are executable files containing collections of functions and classes, which can be used by an application or another DLL at run time. The typical advantage is that if many applications use the same DLL, only one copy needs to be on the disk or loaded in memory, and the size of each executable file will be much smaller. This is what happens with Delphi packages, as well. Chapter 12, "Libraries and Packages," looks at DLLs and packages in more detail.

Rules for Writing Components

Some general rules govern the writing of components. You can find a detailed description of most of them in the *Delphi Component Writer's Guide* Help file, which is required reading for Delphi component writers.

Here is my own summary of the rules for component writers:

- Study the Object Pascal language with care. Particularly important concepts are inheritance, method overriding and overloading, the difference between public and published sections of a class, and the definition of properties and events. If you don't feel confident with the Object Pascal language or the basic ideas about VCL, you can refer to the overall description of the language and library presented in Part I of the book, particularly Chapters 3 ("The Object Pascal Language: Inheritance and Polymorphism") and 5 ("Core Library Classes").
- Study the structure of the VCL class hierarchy and keep a graph of the classes at hand (such as the one included with Delphi).
- Follow the standard Delphi naming conventions. There are several of them for components, as we will see, and following these rules makes it easier for other programmers to interact with your components and further extend them.
- Keep components simple, mimic other components, and avoid dependencies. These three rules basically mean that a programmer using your components should be able to use them as easily as preinstalled Delphi components. Use similar property, method, and event names whenever possible. If users don't need to learn complex rules about the use of your component (that is, if the dependencies between methods or properties are limited) and can simply access properties with meaningful names, they'll be happy.

- Use exceptions. When something goes wrong, the component should raise an exception. When you are allocating resources of any kind, you must protect them with `try/finally` blocks and destructor calls.
- To complete a component, add a bitmap to it, to be used by Delphi's Component Palette. If you intend your component to be used by more than a few people, consider adding a Help file as well.
- Be ready to write *real* code and forget about the visual aspects of Delphi. Writing components generally means writing code without visual support (although Class Completion can speed up the coding of plain classes quite a lot). The exception to this rule is that you can use frames to write components visually.

NOTE You can also use a third-party component writing tool to build your component or to speed up its development. The most powerful third-party tool for creating Delphi components I know of is the Component Development Kit (CDK) from Eagle Software (`www.eagle-software.com`), but many others are available.

The Base Component Classes

To build a new component you generally start from an existing one, or from one of the base classes of VCL. In both cases your component is in one of three broad categories of components (introduced in Chapter 5), set by the three basic classes of the component hierarchy:

- `TWinControl` is the parent class of any component based on a window. Components that descend from this class can receive the input focus and get Windows messages from the system. You can also use their window handle when calling API functions. When creating a brand-new window control, you'll generally inherit from the derived class `TCustomControl`, which has a few extra useful features (particularly some support for painting the control).
- `TGraphicControl` is the parent class of visible components that have no Windows handle (which saves some Windows resources). These components cannot receive the input focus or respond to Windows messages directly. When creating a brand-new graphical control, you'll inherit directly from this class (which has a set of features very similar to `TCustomControl`).
- `TComponent` is the parent class of all components (including the controls) and can be used as a direct parent class for nonvisual components.

In the rest of the chapter, we will build some components using various parent classes, and we'll look at the differences among them. We'll start with components inheriting from existing components or classes at a low level of the hierarchy, and then we'll see examples of classes inheriting directly from the ancestor classes mentioned above.

Building Your First Component

Building components is an important activity for Delphi programmers. The basic idea is that any time you need the same behavior in two different places in an application, or in two different applications, you can place the shared code inside a class—or, even better, a component.

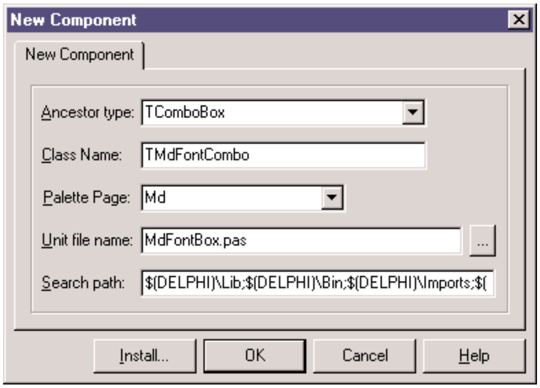
In this section I'll just introduce a couple of simple components, to give you an idea of the steps required to build one and to show you different things you can do to customize an existing component with a limited amount of code.

The Fonts Combo Box

Many applications have a toolbar with a combo box you can use to select a font. If you often use a customized combo box like this, why not turn it into a component? It would probably take less than a minute. To begin, close any active projects in the Delphi environment and start the Component Wizard, either by choosing Component ➢ New Component or by selecting File ➢ New to open the Object Repository and then choosing the Component in the New page. As you can see in Figure 11.1, the Component Wizard requires the following information:

FIGURE 11.1:

Defining the new TMdFontCombo component with the Component Wizard



New Component

Ancestor type: TComboBox

Class Name: TMdFontCombo

Palette Page: Md

Unit file name: MdFontBox.pas

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Imports;$(

- The name of the ancestor type: the component class you want to inherit from. In this case we can use `TComboBox`.
- The name of the class of the new component you are building; we can use `TMdFontCombo`.
- The page of the Component Palette where you want to display the new component, which can be a new or an existing page. We can create a new page, called *Md*.

- The filename of the Pascal unit where Delphi will place the source code of the new component; we can type MdFontBox.
- The current search path (which should be set up automatically).

Click the OK button, and the Component Wizard will generate the following simple Pascal source file with the structure of your component. The Install button can be used to install the component in a package immediately. Let's look at the code first, Listing 11.1, and then discuss the installation.

Listing 11.1: Code of the TMdFontCombo, generated by the Component Wizard

```
unit MdFontBox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TMdFontCombo = class (TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md', [TMdFontCombo]);
end;

end.
```

One of the key elements of this listing is the class definition, which begins by indicating the parent class. The only other relevant portion is the Register procedure. In fact, you can see that the Component Wizard does very little work.

WARNING Starting with Delphi 4, the `Register` procedure *must* be written with an uppercase *R*. This requirement is apparently imposed for C++Builder compatibility (identifiers in C++ are case-sensitive).

TIP Use a naming convention when building components. All the components installed in Delphi should have different class names. For this reason most Delphi component developers have chosen to add a two- or three-letter signature prefix to the names of their components. I've done the same, using *Md* (for *Mastering Delphi*) to identify components built in this book. The advantage of this approach is that you can install my `TMdFontCombo` component even if you've already installed a component named `TFontCombo`. Notice that the unit names must also be unique for all the components installed in the system, so I've applied the same prefix to the unit names.

That's all it takes to build a component. Of course, in this example there isn't a lot of code. We need only copy all the system fonts to the `Items` property of the combo box at startup. To accomplish this, we might try to override the `Create` method in the class declaration, adding the statement `Items := Screen.Fonts`. However, this is not the correct approach. The problem is that we cannot access the combo box's `Items` property before the window handle of the component is available; the component cannot have a window handle until its `Parent` property is set; and that property isn't set in the constructor, but later on.

For this reason, instead of assigning the new strings in the `Create` constructor, we must perform this operation in the `CreateWnd` procedure, which is called to create the window control after the component is constructed, its `Parent` property is set, and its window handle is available. Again, we execute the default behavior, and then we can write our custom code. I could have skipped the `Create` constructor and written all the code in `CreateWnd`, but I decided to use both startup methods to demonstrate the difference between them. Here is the declaration of the component class:

```
type
  TMdFontCombo = class (TComboBox)
  private
    FChangeFormFont: Boolean;
    procedure SetChangeFormFont(const Value: Boolean);
  public
    constructor Create (AOwner: TComponent); override;
    procedure CreateWnd; override;
    procedure Change; override;
  published
    property Style default csDropDownList;
    property Items stored False;
    property ChangeFormFont: Boolean
      read FChangeFormFont write SetChangeFormFont default True;
  end;
```

And here is the source code of its two methods executed at startup:

```
constructor TMdFontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
  FChangeFormFont := True;
end;

procedure TMdFontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items.Assign (Screen.Fonts);

  // grab the default font of the owner form
  if FChangeFormFont and Assigned (Owner) and (Owner is TForm) then
    ItemIndex := Items.IndexOf ((Owner as TForm).Font.Name);
end;
```

Notice that besides giving a new value to the component's `Style` property, in the `Create` method, I've redefined this property by setting a value with the `default` keyword. We have to do both operations because adding the `default` keyword to a property declaration has no direct effect on the property's initial value. Why specify a property's default value then? Because properties that have a value equal to the default are not streamed with the form definition (and they don't appear in the textual description of the form, the DFM file). The `default` keyword tells the streaming code that the component initialization code will set the value of that property.

TIP Why is it important to specify a default value for a published property? To reduce the size of the DFM files and, ultimately, the size of the executable files (which include the DFM files).

The other redefined property, `Items`, is set as a property that should not be saved to the DFM file at all, regardless of the actual value. This is obtained with the `stored` directive followed by the value False. The component and its window are going to be created again when the program starts, so it doesn't make any sense to save in the DFM file information that will be discarded later on (to be replaced with the new list of fonts).

NOTE We could have even written the code of the `CreateWnd` method to copy the fonts to the combo box items only at run time. This can be done by using statements such as `if not (csDesigning in ComponentState) then`…. But for this first component we are building, the less efficient but more straightforward method described above offers a clearer illustration of the basic procedure.

The third property, ChangeFormFont, is not inherited but introduced by the component. It is used to determine whether the current font selection of the combo box should determine the font of the form hosting the component. Again this property is declared with a default value, set in the constructor. The ChangeFormFont property is used in the code of the CreateWnd method, shown before, to set up the initial selection of the combo depending on the font of the form hosting the component. This is generally the Owner of the component, although I could have also walked the Parent tree looking for a form component. This code isn't perfect, but the Assigned and is tests provide some extra safety.

The ChangeFormFont property and the same if test play a key role in the Changed method, which in the base class triggers the OnChange event. By overriding this method we provide a default behavior, which can be disabled by toggling the value of the property, but also allow the execution of the OnChange event, so that users of this class can fully customize its behavior. The final method, SetChangeFormFont, has been modified to refresh the form's font in case the property is being turned on. This is the complete code:

```
procedure TMdFontCombo.Change;
begin
  // assign the font to the owner form
  if FChangeFormFont and Assigned (Owner) and (Owner is TForm) then
    TForm (Owner).Font.Name := Text;
  inherited;
end;

procedure TMdFontCombo.SetChangeFormFont(const Value: Boolean);
begin
  FChangeFormFont := Value;
  // refresh font
  if FChangeFormFont then
    Change;
end;
```

Creating a Package

Now we have to install the component in the environment, using a package. For this example, we can either create a new package or use an existing one, like the default user's package.

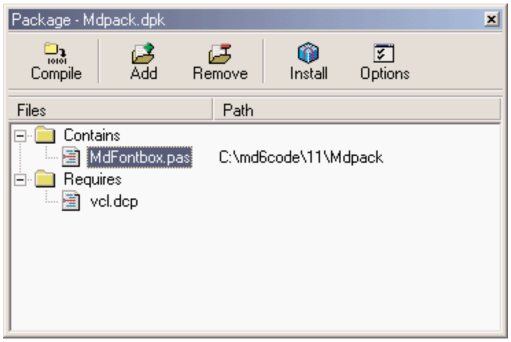
In each case, choose the Component ➢ Install Component menu command. The resulting dialog box has a page to install the component into an existing package, and a page to create a new package. In this last case, simply type in a filename and a description for the package. Clicking OK opens the Package Editor (see Figure 11.2), which has two parts:

- The Contains list indicates the components included in the package (or, to be more precise, the units defining those components).

- The `Requires` list indicates the packages required by this package. Your package will generally require the rtl and vcl packages (the main run-time library package and core VCL package), but it might also need the vcldb package (which includes most of the database-related classes) if the components of the new package do any database-related operations.

FIGURE 11.2:

The Package Editor



NOTE Package names in Delphi 6 aren't version specific any more, even if the compiled packages still have a version number in the filename. See the section "Project and Library Names in Delphi 6" in Chapter 12, "Libraries and Packages," for more details on how this is technically achieved.

If you add the component to the new package we've just defined, and then simply compile the package and install it (using the two corresponding toolbar buttons of the package editor), you'll immediately see the new component show up in the Md page of the Component Palette. The `Register` procedure of the component unit file told Delphi where to install the new component. By default, the bitmap used will be the same as the parent class, because we haven't provided a custom bitmap (we will do this in later examples). Notice also that if you move the mouse over the new component, Delphi will display as a hint the name of the class without the initial letter *T*.

What's Behind a Package?

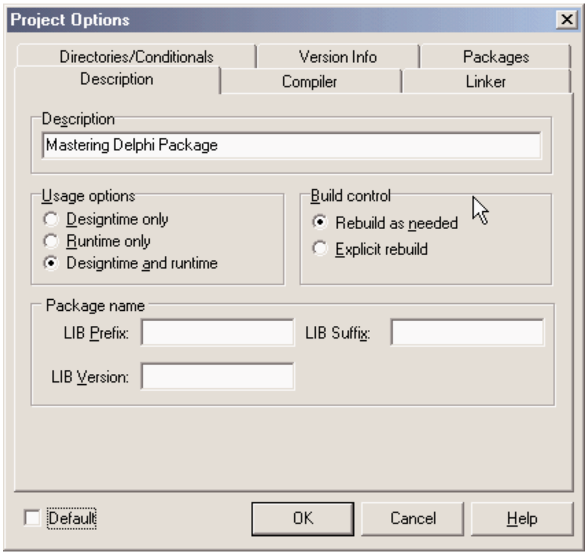
What is behind the package we've just built? The Package Editor basically generates the source code for the package project: a special kind of DLL built in Delphi. The package project is saved in a file with the DPK (for Delphi PacKage) extension. A typical package project looks like this:

```
package MdPack;

{$R *.RES}
```

```
{$ALIGN ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
...
{$DESCRIPTION 'Mastering Delphi Package'}
{$IMPLICITBUILD ON}

requires
  vcl;

contains
  MdFontBox in 'MdFontBox.pas';

end.
```

As you can see, Delphi uses specific language keywords for packages: the first is the `package` keyword (which is similar to the `library` keyword I'll discuss in the next chapter). This keyword introduces a new package project. Then comes a list with all the compiler options, some of which I've omitted from the listing. Usually the options for a Delphi project are stored in a separate file; packages, by contrast, include all the compiler options directly in their source code. Among the compiler options there is a `DESCRIPTION` compiler directive, used to make the package description available to the Delphi environment. In fact, after you've installed a new package, its description will be shown in the Packages page of the Project Options dialog box, a page you can also activate by selecting the Component ➢ Install Packages menu item. This dialog box is shown in Figure 11.3.

FIGURE 11.3: The Project Options for packages. You can see the new package we've just created.



Description: Mastering Delphi Package

Usage options:
○ Designtime only
○ Runtime only
● Designtime and runtime

Build control:
● Rebuild as needed
○ Explicit rebuild

LIB Prefix:
LIB Suffix:
LIB Version:

□ Default

Besides common directives like the DESCRIPTION one, there are other compiler directives specific to packages. The most common of these options are easily accessible through the Options button of the Package Editor. After this list of options come the `requires` and `contains` keywords, which list the items displayed visually in the two pages of the Package Editor. Again, the first is the list of packages required by the current one, and the second is a list of the units installed by this package.

What is the technical effect of building a package? Besides the DPK file with the source code, Delphi generates a BPL file with the dynamic link version of the package and a DCP file with the symbol information. In practice, this DCP file is the sum of the symbol information of the DCU files of the units contained in the package.

At design time, Delphi requires both the BPL and DCP files, because the first has the actual code of the components created on the design form and the symbol information required by the code insight technology. If you link the package dynamically (using it as a run-time package), the DCP file will also be used by the linker, and the BPL file should be shipped along with the main executable file of the application. If you instead link the package statically, the linker refers to the DCU files, and you'll need to distribute only the final executable file.

For this reason, as a component designer, you should generally distribute at least the BPL file, the DCP file, and the DCU files of the units contained in the package and any corresponding DFM files, plus a Help file. As an option, of course, you might also make available the source code files of the package units (the PAS files) and of the package itself (the DPK file).

WARNING Delphi, by default, will place all the compiled package files (BPL and DCP) not in the folder of the package source code but under the `\Projects\BPL` folder. This is done so that the IDE can easily locate them, and creates no particular problem. When you have to compile a project using components declared on those packages, though, Delphi might complain that it cannot find the corresponding DCU files, which are stored in the package source code folder. This problem can be solved by indicating the package source code folder in the Library Path (in the Environment Options, which affect all projects) or by indicating it in the Search Path of the current project (in the Project Options). If you choose the first approach, placing different components and packages in a single folder might result in a real time-saver.

Installing the Components of This Chapter

Having built our first package, we can now start using the component we've added to it. Before we do so, however, I should mention that I've extended the MdPack package to include all of the components we are going to build in this chapter, including different versions of the same component. I suggest you install this package. The best approach is to copy it into a directory of your path, so that it will be available both to the Delphi environment and to the programs you build with it. I've collected all the component source code files and the package definition in a single subdirectory, called MdPack. This allows the Delphi environment, or a specific project, to refer only to one directory when looking for the DCU files of this package. As suggested in the warning above, I could have collected all of the components presented in the book in a single folder on the companion CD, but I decided that keeping the chapter-based organization was actually more understandable for readers.

Remember, anyway, that if you compile an application using the packages as run-time DLLs, you'll need to install these new libraries on your clients' computers. If you instead compile the programs by statically linking the package, the DLL will be required only by the development environment and not by the users of your applications.

Using the Font Combo Box

Now you can create a new Delphi program to test the Font combo box. Move to the Component Palette, select the new component, and add it to a new form. A traditional-looking combo box will appear. However, if you open the Items property editor, you'll see a list of the fonts installed on your computer. To build a simple example, I've added a Memo component to the form with some text inside it. By leaving the `ChangeFormFont` property on, you don't need to write any other code to the program, as you'll see in the example. As an alternative I could have turned off the property and handled the `OnChange` event of the component, with code like this:

```
Memo1.Font.Name := MdFontCombo1.Text;
```

The aim of this simple program is only to test the behavior of the new component we have built. The component is still not very useful—we could have added a couple of lines of code to a form to obtain the same effect—but looking at a couple of simple components should help you get an idea of what is involved in component building.

Creating Compound Components

The next component I want to focus on is a digital clock. This example has some interesting features. First, it embeds a component (a Timer) in another component; second, it shows the live-data approach.

NOTE The first feature has become even more relevant in Delphi 6, as the Object Inspector of the latest version of Delphi allows you to expose properties of subcomponents directly. As an effect, the example presented in this section has been modified (and simplified) compared to the previous edition of the book. I'll actually mention the differences, when relevant.

Since the digital clock will provide some text output, I considered inheriting from the `TLabel` class. However, this would allow a user to change the label's caption—that is, the text of the clock. To avoid this problem, I simply used the `TCustomLabel` component as the parent class.

A TCustomLabel object has the same capabilities as a TLabel object, but few published properties. In other words, a TCustomLabel subclass can decide which properties should be available and which should remain hidden.

NOTE Most of the Delphi components, particularly the Windows-based ones, have a TCustomXxx base class, which implements the entire functionality but exposes only a limited set of properties. Inheriting from these base classes is the standard way to expose only some of the properties of a component in a customized version. In fact, you cannot hide public or published properties of a base class.

With past versions of Delphi, the component had to define a new property, Active, wrapping the Enabled property of the Timer. A *wrapper* property means that the get and set methods of this property read and write the value of the *wrapped* property, which belongs to an internal component (a wrapper property generally has no local data). In this specific case, the code looked like this:

```
function TMdClock.GetActive: Boolean;
begin
  Result := FTimer.Enabled;
end;

procedure TMdClock.SetActive (Value: Boolean);
begin
  FTimer.Enabled := Value;
end;
```

Publishing Subcomponents in Delphi 6

With Delphi 6 we can simply expose the entire subcomponent, the timer, in a property of its own, that will be regularly expanded by the Object Inspector, allowing a user to set each and every of its subproperties, and even to handle its events.

Here is the full type declaration for the TMdClock component, with the subcomponent declared in the private data and exposed as a published property (in the last line):

```
type
  TMdClock = class (TCustomLabel)
  private
    FTimer: TTimer;
  protected
    procedure UpdateClock (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
  published
    property Align;
    property Alignment;
    property Color;
    property Font;
    property ParentColor;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property Transparent;
    property Visible;
    property Timer: TTimer read FTimer;
  end;
```

The `Timer` property is read-only, as I don't want users to select another value for this component in the Object Inspector (or detach the component by clearing the value of this property). Developing sets of subcomponents that can be used alternately is certainly possible, but adding write support for this property in a safe way is far from trivial (considering that the users of your component might not be very expert Delphi programmers). So I suggest you to stick with read-only properties for subcomponents.

To create the Timer, we must override the constructor of the clock component. The `Create` method calls the corresponding method of the base class and creates the Timer object, installing a handler for its `OnTimer` event:

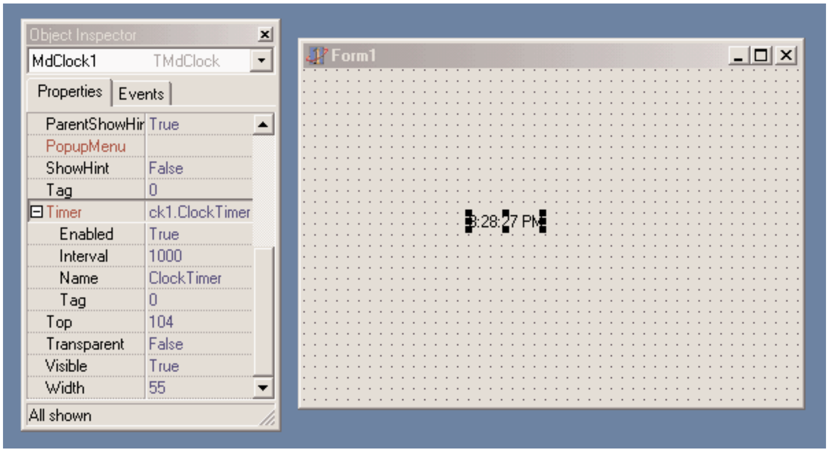
```
constructor TMdClock.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  // create the internal timer object
  FTimer := TTimer.Create (Self);

  FTimer.Name := 'ClockTimer';
  FTimer.OnTimer := UpdateClock;
  FTimer.Enabled := True;
  FTimer.SetSubComponent (True);
end;
```

The code gives the component a name, for display in the Object Inspector (see Figure 11.4), and calls the specific `SetSubComponent` method. We don't need a destructor, simply because the `FTimer` object has our `TMDClock` component as owner (as indicated by the parameter of its `Create` constructor), so it will be destroyed automatically when the clock component is destroyed.

NOTE What is the actual effect of the call to the `SetSubComponent` method in the code above? This call sets an internal flag, saved in the `ComponentStyle` property set. The flag (csSubComponent) affects the streaming system, allowing the subcomponent and its properties to be saved in the DFM file. In fact, the streaming system by default ignores components that are not owned by the form.

FIGURE 11.4:

In Delphi 6, the Object Inspector can automatically expand subcomponents, showing their properties, as in the case of the Timer property of the MdLabelClock component.



Object Inspector

MdClock1 TMdClock

Properties | Events

ParentShowHir	True
PopupMenu	
ShowHint	False
Tag	0
⊟ Timer	ck1.ClockTimer
Enabled	True
Interval	1000
Name	ClockTimer
Tag	0
Top	104
Transparent	False
Visible	True
Width	55

All shown

The key piece of the component's code is the `UpdateClock` procedure, which is just one statement:

```
procedure TMdLabelClock.UpdateClock (Sender: TObject);
begin
  // set the current time as caption
  Caption := TimeToStr (Time);
end;
```

This method uses `Caption`, which is an unpublished property, so that a user of the component cannot modify it in the Object Inspector. The result of this statement is to display the current time. This happens continuously, because the method is connected to the Timer's `OnTimer` event.

The Component Palette Bitmaps

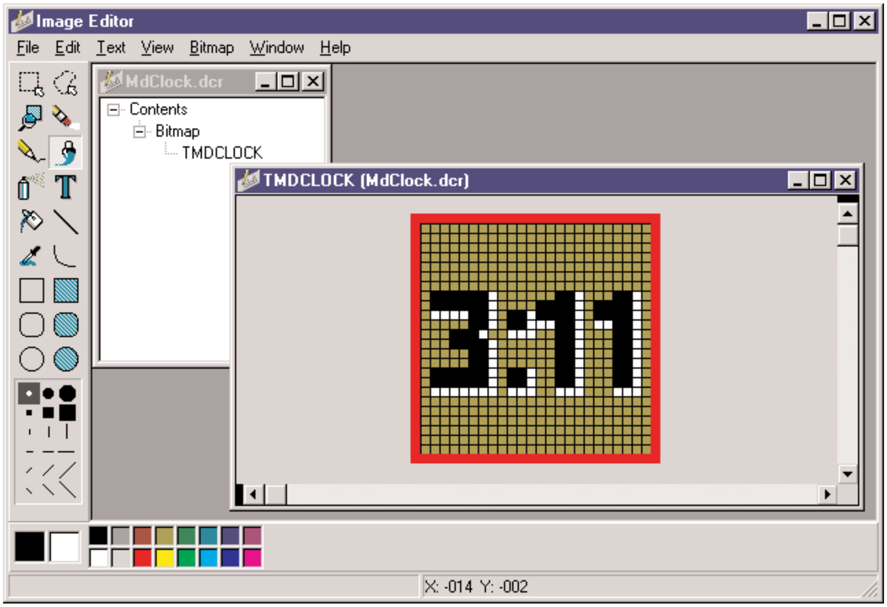
Before installing this second component, we can take one further step: define a bitmap for the Component Palette. If we fail to do so, the Palette uses the bitmap of the parent class, or a default object's bitmap if the parent class is not an installed component (as is the case of the `TCustomLabel`). Defining a new bitmap for the component is easy, once you know the rules. You can create one with the Image Editor (as shown in Figure 11.5), starting a new project and selecting the Delphi Component Resource (DCR) project type.

TIP

DCR files are simply standard RES files with a different extension. If you prefer, you can create them with any resource editor, including the Borland Resource Workshop, which is certainly a more powerful tool than the Delphi Image editor. When you finish creating the resource file, simply rename the RES file to use a DCR extension.

FIGURE 11.5:

The definition of a Component Palette bitmap in Delphi's Image Editor



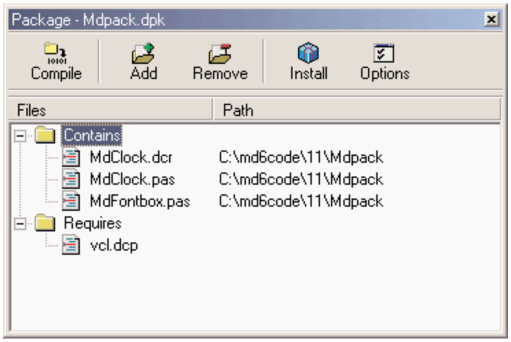
Now we can add a new bitmap to the resource, choosing a size of 24×24 pixels, and we are ready to draw the bitmap. The other important rules refer to naming. In this case, the naming rule is not just a convention; it is a requirement so that the IDE can find the image for a given component class:

- The name of the bitmap resource must match the name of the component, including the initial *T*. In this case, the name of the bitmap resource should be TMDCLOCK. The name of the bitmap resource must be uppercase—this is mandatory.
- If you want the Package Editor to recognize and include the resource file, the name of the DCR file must match the name of the compiled unit that defines the component. In this case, the filename should be `MdClock.DCR`. If you manually include the resource file, via a `$R` directive, you can give it the name you like, and also use a RES or DCR file with multiple palette icons.

When the bitmap for the component is ready, you can install the component in Delphi, by using the Package Editor's Install Package toolbar button. After this operation, the Contains section of the editor should list both the PAS file of the component and the corresponding DCR file. In Figure 11.6 you can see all the files (including the DCR files) of the final version of the MdPack package. If the DCR installation doesn't work properly, you can manually add the `{$R unitname.dcr}` statement in the package source code.

FIGURE 11.6:

The Contains section of the Package Editor shows both the units that are included in the package and the component resource files.



Files	Path
Contains	
MdClock.dcr	C:\md6code\11\Mdpack
MdClock.pas	C:\md6code\11\Mdpack
MdFontbox.pas	C:\md6code\11\Mdpack
Requires	
vcl.dcp	

Building Compound Components with Frames

Instead of building the compound component in code and hooking up the timer event manually, we could have obtained a similar effect by using a frame. Frames make the development of compound components with custom event handlers a visual operation, and thus simpler. You can share this frame by adding it to the Repository or by creating a template using the Add to Palette command of the frame's shortcut menu.

As an alternative, you might want to share the frame by placing it in a package and registering it as a component. Technically, this is not difficult. You add a `Register` procedure to the frame's unit, add the unit to a package, and build it. The new component/frame will be in the Component Palette, like any other component. In Delphi 6, when you place this component/frame on a form, you'll see its subcomponents. You cannot select these subcomponents with a mouse click in the Form Designer, but can do it in the Object TreeView. However, any change you make to these components at design time will inevitably get lost when you run the program or save and reload the form, because the changes to those subcomponents won't be streamed, contrary to what happened with standard frames you place inside a form.

If this is not what you might expect, I've found a *reasonable* way to use frames in packages, demonstrated by the MdFramedClock component, part of the examples on the CD for this chapter. The idea is to turn the components owned by the form into actual subcomponents, by calling the new `SetSubComponent` method. As I was up to it, I've also exposed the internal components with properties, even if this isn't compulsory as they can be selected in the Object TreeView anyway. This is the declaration of the component and the code of its methods:

```
type
  TMdFramedClock = class(TFrame)
    Label1: TLabel;
    Timer1: TTimer;
    Bevel1: TBevel;
```

Continued on next page

```
    procedure Timer1Timer(Sender: TObject);
  public
    constructor Create(AOnwer: TComponent); override;
  published
    property SubLabel: TLabel read Label1;
    property SubTimer: TTimer read Timer1;
  end;

constructor TMdFramedClock.Create(AOnwer: TComponent);
begin
  inherited;
  Timer1.SetSubComponent (true);
  Label1.SetSubComponent (true);
end;

procedure TMdFramedClock.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr (Time);
end;
```

In contrast to the clock component built earlier, there is no need to set up the properties of the timer, or to connect the timer event to its handler function manually, as this is done visually and saved in the DFM file of the frame. Notice also that I haven't exposed the Bevel component—I haven't called `SetSubComponent` on it—so that you can try editing it at design time and see that all the changes get lost, as I mentioned above.

After you install this frame/component, you can use it inside any application. In this particular case, as soon as you drop the frame on the form, the timer will start to update the label with the current time. However, you can still handle its `OnTimer` event, and the Delphi IDE (recognizing that the component is inside a frame) will define a method with this predefined code:

```
procedure TForm1.MdFramedClock1Timer1Timer(Sender: TObject);
begin
  MdFramedClock1.Timer1Timer(Sender);
end;
```

As soon as this timer is connected, even at design time, the live clock will stop, as its original event handler is disconnected. After compiling and running the program, however, the original behavior will be restored (at least if you don't delete the line above) and your extra custom code will be executed as well. This is exactly what you'll expect from frames. You can find a complete demo of the use of this frame/component in the FrameClock example.

Continued on next page

As a short conclusion of this digression on frames compiled inside packages, I can certainly say that this approach is still far from linear. It is certainly much better than in Delphi 5, where frames inside packages were really unusable. The question is, is it worth the effort? In short, I'd say no. If you work alone or with a small team, it's better to use plain frames stored in the Repository. In larger organizations and for distributing your frames to a larger audience, I bet most people will rather build their components in the traditional way, without trying to use frames. In other words, I'm still hoping that Borland will address more complete support to the visual development of packaged components based on frames.

A Complex Graphical Component

The graphical component I want to build is an arrow component. You can use such a component to indicate a flow of information, or an action, for example. This component is quite complex, so I'll show you the various steps instead of looking directly at the complete source code. The component I've added to the MdPack package on the CD is only the final version of this process, which will demonstrate several important concepts:

- The definition of new enumerated properties, based on custom enumerated data types.
- The use of properties of `TPersistent`-derived classes, such as `TPen` and `TBrush`, and the issues related to their creation and destruction, and to handling their `OnChange` events internally in our component.
- The implementation of the `Paint` method of the component, which provides its user interface and should be generic enough to accommodate all the possible values of the various properties, including its `Width` and `Height`. The `Paint` method plays a substantial role in this graphical component.
- The definition of a custom event handler for the component, responding to user input (in this case, a double-click on the point of the arrow). This will require direct handling of Windows messages and the use of the Windows API for graphic regions.
- The registration of properties in Object Inspector categories and the definition of a custom category.

Defining an Enumerated Property

After generating the new component with the Component Wizard and choosing `TGraphicControl` as the parent class, we can start to customize the component. The arrow can point in any of four directions: up, down, left, or right. An enumerated type expresses these choices:

```
type
  TMdArrowDir = (adUp, adRight, adDown, adLeft);
```

This enumerated type defines a private data member of the component, a parameter of the procedure used to change it, and the type of the corresponding property. Two more simple properties are `ArrowHeight` and `Filled`, the first determining the size of the arrowhead and the second whether to fill the arrowhead with color:

```
type
  TMdArrow = class (TGraphicControl)
  private
    fDirection: TMdArrowDir;
    fArrowHeight: Integer;
    fFilled: Boolean;
    procedure SetDirection (Value: TMd4ArrowDir);
    procedure SetArrowHeight (Value: Integer);
    procedure SetFilled (Value: Boolean);
  published
    property Width default 50;
    property Height default 20;
    property Direction: TMd4ArrowDir
      read fDirection write SetDirection default adRight;
    property ArrowHeight: Integer
      read fArrowHeight write SetArrowHeight default 10;
    property Filled: Boolean read fFilled write SetFilled default False;
```

NOTE A graphic control has no default size, so when you place it in a form, its size will be a single pixel. For this reason it is important to add a default value for the `Width` and `Height` properties and set the class fields to the default property values in the constructor of the class.

The three custom properties are read directly from the corresponding field and are written using three `Set` methods, all having the same standard structure:

```
procedure TMdArrow.SetDirection (Value: TMdArrowDir);
begin
  if fDirection <> Value then
  begin
    fDirection := Value;
    ComputePoints;
    Invalidate;
  end;
end;
```

Notice that we ask the system to repaint the component (by calling `Invalidate`) only if the property is really changing its value and after calling the `ComputePoints` method, which computes the triangle delimiting the arrowhead. Otherwise, the code is skipped and the method ends immediately. This code structure is very common, and we will use it for most of the `Set` procedures of properties.

We must also remember to set the default values of the properties in the component's constructor:

```
constructor TMdArrow.Create (AOwner: TComponent);
begin
  // call the parent constructor
  inherited Create (AOwner);
  // set the default values
  fDirection := adRight;
  Width := 50;
  Height := 20;
  fArrowHeight := 10;
  fFilled := False;
```

In fact, as mentioned before, the default value specified in the property declaration is used only to determine whether to save the property's value to disk. The `Create` constructor is defined in the public section of the type definition of the new component, and it is indicated by the `override` keyword. It is fundamental to remember this keyword; otherwise, when Delphi creates a new component of this class, it will call the constructor of the base class, rather than the one you've written for your derived class.

Property-Naming Conventions

In the definition of the Arrow component, notice the use of several naming conventions for properties, access methods, and fields. Here is a summary:

- A property should have a meaningful and readable name.
- When a private data field is used to hold the value of a property, the field should be named with an *f* (field) at the beginning, followed by the name of the corresponding property.
- When a function is used to change the value of the property, the function should have the word *Set* at the beginning, followed by the name of the corresponding property.
- A corresponding function used to read the property should have the word *Get* at the beginning, again followed by the property name.

These are just guidelines to make programs more readable. The compiler doesn't enforce them. These conventions are described in the *Delphi Component Writers' Guide* and are followed by the Delphi's class completion mechanism.

Writing the *Paint* Method

Drawing the arrow in the various directions and with the various styles requires a fair amount of code. To perform custom painting, you override the `Paint` method and use the protected `Canvas` property.

Instead of computing the position of the arrowhead points in drawing code that will be executed often, I've written a separate function to compute the arrowhead area and store it in an array of points defined among the private fields of the component as:

```
fArrowPoints: array [0..3] of TPoint;
```

These points are determined by the `ComputePoints` private method, which is called every time some of the component properties change. Here is an excerpt of its code:

```
procedure TMdArrow.ComputePoints;
var
  XCenter, YCenter: Integer;
begin
  // compute the points of the arrowhead
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  case FDirection of
    adUp: begin
      fArrowPoints [0] := Point (0, FArrowHeight);
      fArrowPoints [1] := Point (XCenter, 0);
      fArrowPoints [2] := Point (Width-1, FArrowHeight);
    end;
  // and so on for the other directions
```

The code computes the center of the component area (simply dividing the `Height` and `Width` properties by two) and then uses it to determine the position of the arrowhead. Besides changing the direction or other properties, we need to refresh the position of the arrowhead when the size of the component changes. What we can do is to override the `SetBounds` method of the component, which is called by VCL every time the `Left`, `Top`, `Width`, and `Height` properties of a component change:

```
procedure TMdArrow.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited SetBounds (ALeft, ATop, AWidth, AHeight);
  ComputePoints;
end;
```

Once the component knows the position of the arrowhead, its painting code becomes simpler. Here is an excerpt of the `Paint` method:

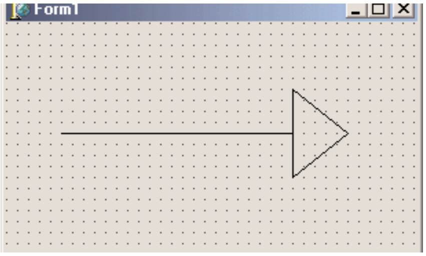
```
procedure TMdArrow.Paint;
var
  XCenter, YCenter: Integer;
```

```
begin
  // compute the center
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;

  // draw the arrow line
  case FDirection of
    adUp: begin
      Canvas.MoveTo (XCenter, Height-1);
      Canvas.LineTo (XCenter, FArrowHeight);
    end;
    // and so on for the other directions
  end;

  // draw the arrow point, eventually filling it
  if FFilled then
    Canvas.Polygon (fArrowPoints)
  else
    Canvas.PolyLine (fArrowPoints);
end;
```

You can see an example of the output of this component in Figure 11.7.

FIGURE 11.7:

The output of the Arrow component



Adding *TPersistent* Properties

To make the output of the component more flexible, I've added to it two new properties, defined with a class type (specifically, a `TPersistent` data type, which defines objects that can be automatically streamed by Delphi). These properties are a little more complex to handle, because the component now has to create and destroy these internal objects (as we did with the internal Timer of the clock component). This time, however, we also export the internal objects using some properties, so that users can directly change them from the Object Inspector. To update the component when these subobjects change, we'll also need to handle their internal OnChange property. Here is the definition of the two new TPersistent-type properties and the other changes to the definition of the component class:

```
type
  TMdArrow = class (TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    ...
    procedure SetPen (Value: TPen);
    procedure SetBrush (Value: TBrush);
    procedure RepaintRequest (Sender: TObject);
  published
    property Pen: TPen read FPen write SetPen;
    property Brush: TBrush read FBrush write SetBrush;
  end;
```

The first thing to do is to create the objects in the constructor and set their OnChange event handler:

```
constructor TMdArrow.Create (AOwner: TComponent);
begin
  ...
  // create the pen and the brush
  FPen := TPen.Create;
  FBrush := TBrush.Create;
  // set a handler for the OnChange event
  FPen.OnChange := RepaintRequest;
  FBrush.OnChange := RepaintRequest;
end;
```

These OnChange events are fired when one of the properties of these subobjects changes; all we have to do is to ask the system to repaint our component:

```
procedure TMdArrow.RepaintRequest (Sender: TObject);
begin
  Invalidate;
end;
```

You must also add to the component a destructor, to remove the two graphical objects from memory (and free their system resources):

```
destructor TMdArrow.Destroy;
begin
  FPen.Free;
  FBrush.Free;
  inherited Destroy;
end;
```

The properties related to these two components require some special handling: instead of copying the pointer to the objects, we should copy the internal data of the object passed as parameter. The standard := operation copies the pointers, so in this case we have to use the Assign method instead. Here is one of the two Set procedures:

```
procedure TMdArrow.SetPen (Value: TPen);
begin
  FPen.Assign(Value);
  Invalidate;
end;
```

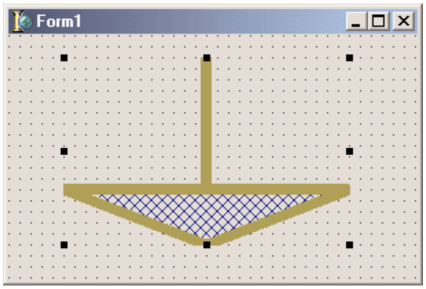
Many TPersistent classes have an Assign method that should be used when we need to update the data of these objects. Now, to actually use the pen and brush for the drawing, you have to modify the Paint method, setting the Pen and the Brush properties of the component Canvas to the value of the internal objects before drawing any line:

```
procedure TMdArrow.Paint;
begin
  // use the current pen and brush
  Canvas.Pen := FPen;
  Canvas.Brush := FBrush;
```

You can see an example of the new output of the component in Figure 11.8.

FIGURE 11.8:

The output of the Arrow component with a thick pen and a special hatch brush



Defining a New Custom Event

To complete the development of the Arrow component, let's add a custom event. Most of the time, new components use the events of their parent classes. For example, in this component, I've made some standard events available simply by redeclaring them in the published section of the class:

```
type
  TMdArrow = class (TGraphicControl)
  published
    property OnClick;
```

```
        property OnDragDrop;
        property OnDragOver;
        property OnEndDrag;
        property OnMouseDown;
        property OnMouseMove;
        property OnMouseUp;
```

Thanks to this declaration, the above events (originally declared in a parent class) will now be available in the Object Inspector when the component is installed.

Sometimes, however, a component requires a custom event. To define a brand-new event, you first need to add to the class a field of the type of the event. This type is actually a method pointer type (see Chapter 5 for details). Here is the definition I've added in the private section of the TMdArrow class:

```
fArrowDblClick: TNotifyEvent;
```

In this case I've used the TNotifyEvent type, which has only a Sender parameter and is used by Delphi for many events, including OnClick and OnDblClick events. Using this field I've defined a very simple published property, with direct access to the field:

```
property OnArrowDblClick: TNotifyEvent
  read fArrowDblClick write fArrowDblClick;
```

Notice again the standard naming convention, with event names starting with *On*. The fArrowDblClick method pointer is activated (executing the corresponding function) inside the specific ArrowDblClick dynamic method. This happens only if an event handler has been specified in the program that uses the component:

```
procedure TMdArrow.ArrowDblClick;
begin
  if Assigned (FArrowDblClick) then
    FArrowDblClick (Self);
end;
```

This method is defined in the protected section of the type definition to allow future subclasses to both call and change it. Basically, the ArrowDblClick method is called by the handler of the wm_LButtonDblClk Windows message, but only if the double-click took place inside the arrow's point. To test this condition, we can use some of the Windows API's region functions.

NOTE A *region* is an area of the screen enclosed by any shape. For example, we can build a polygonal region using the three vertices of the arrow-point triangle. The only problem is that to fill the surface properly, we must define an array of TPoints in a clockwise direction (see the description of the CreatePolygonalRgn in the Windows API Help for the details of this strange approach). That's what I did in the ComputePoints method.

Once we have defined a region, we can test whether the point where the double-click occurred is inside the region by using the PtInRegion API call. You can see the complete source code of this procedure in the following listing:

```
procedure TMdArrow.WMLButtonDblClk (
  var Msg: TWMLButtonDblClk); // message wm_LButtonDblClk;
var
  HRegion: HRgn;
begin
  // perform default handling
  inherited;

  // compute the arrowhead region
  HRegion := CreatePolygonRgn (fArrowPoints, 3, WINDING);
  try  // check whether the click took place in the region
    if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then
      ArrowDblClick;
  finally
    DeleteObject (HRegion);
  end;
end;
```

Registering Property Categories

We've added to this component some custom properties and a new event. If you arrange the properties in the Object Inspector by category (a feature available since Delphi 5), all the new elements will show up in the generic Miscellaneous category. Of course, this is far from ideal, but we can easily register the new properties in one of the available categories.

We can register a property (or an event) in a category by calling one of the four overloaded versions of the RegisterPropertyInCategory function, defined in the new DesignIntf unit. When calling this function, you indicate the name of the category, and you can specify the property name, its type, or the property name and the component it belongs to. For example, we can add the following lines to the Register procedure of the unit to register the OnArrowDblClick event in the Input category and the Filled property in the Visual category:

```
uses
  DesignIntf;

procedure Register;
begin
  RegisterComponents('Md', [TMdArrow]);
  RegisterPropertyInCategory ('Input', TMdArrow, 'OnArrowDblClick');
  RegisterPropertyInCategory ('Visual', TMdArrow, 'Filled');
end;
```

In Delphi 5, the first parameter was a class indicating the category type; now the parameter is simply a string, a much simpler solution. This change also makes it straightforward to define new categories: you simply pass its name as the first parameter of the RegisterPropertyInCategory function, as in:

```
RegisterPropertyInCategory ('Arrow', TMdArrow, 'Direction');
RegisterPropertyInCategory ('Arrow', TMdArrow, 'ArrowHeight');
```

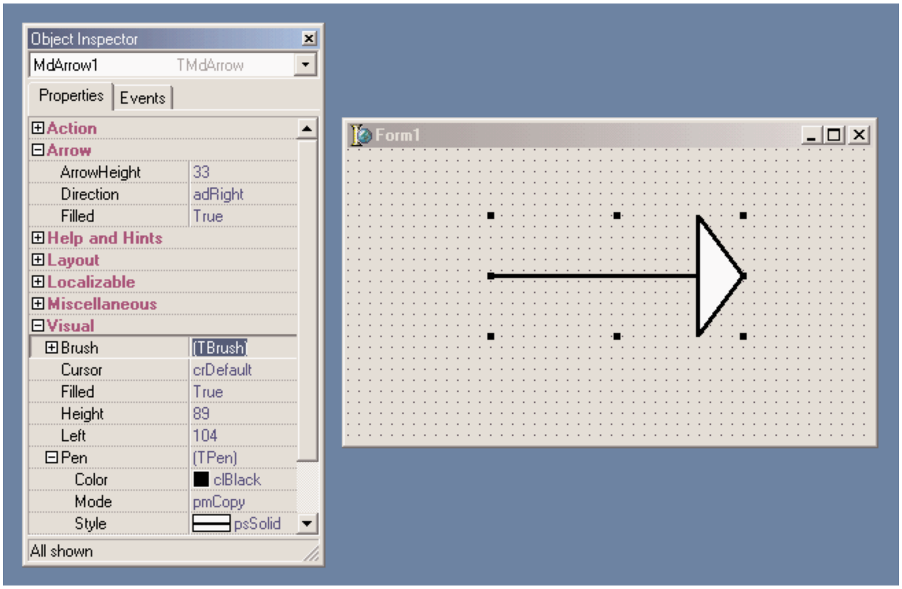
Creating a brand new category for the specific properties of our component can make it much simpler for a user to locate its specific features. Notice, though, that since we rely on the DesignIntf unit, you should compile the unit containing these registrations in a design-time package, not a run-time one (in fact, the required DesignIde unit cannot be distributed). For this reason, I've written this code in a separate unit than the one defining the component and added the new unit (MdArrReg) to the package MdDesPk, including all of the design-time-only units; this is discussed later, in the section "Installing the Property Editor."

WARNING It's debatable whether using a category for the specific properties of a component is a good idea. On one side, a user of the component can easily spot specific properties. At the same time, some of the new properties might not pertain to any of the existing categories. On the other side, however, categories can be overused. If every component introduces new categories, users may get confused. You also face the risk of having as many categories as there are properties.

Notice that my code registers the Filled property in two different categories. This is not a problem, because the same property can show up multiple times in the Object Inspector under different groups, as you can see in Figure 11.9.

To test the arrow component I've written a very simple example program, ArrowDemo, which allows you to modify most of its properties at run time. This type of test, after you have written a component or while you are writing it, is very important.

NOTE The *Localizable* property category has a special role, related to the use of the ITE (Integrated Translation Environment). When a property is part of this category, its value will be listed in the ITE as a property that can be translated into another language. (A complete discussion of the ITE is beyond the scope of this book.)



Object Inspector

MdArrow1 TMdArrow

Properties | Events

⊞Action	
⊟Arrow	
ArrowHeight	33
Direction	adRight
Filled	True
⊞Help and Hints	
⊞Layout	
⊞Localizable	
⊞Miscellaneous	
⊟Visual	
⊞Brush	(TBrush)
Cursor	crDefault
Filled	True
Height	89
Left	104
⊟Pen	(TPen)
Color	■ clBlack
Mode	pmCopy
Style	psSolid

All shown

FIGURE 11.9: The Arrow component defines a custom property category, *Arrow*, as you can see in the Object Inspector.

Customizing Windows Controls

One of the most common ways of customizing existing components is to add some predefined behavior to their event handlers. Every time you need to attach the same event handler to components of different forms, you should consider adding the code of the event right into a subclass of the component. An obvious example is that of edit boxes accepting only numeric input. Instead of attaching to each of them a common `OnChar` event handler, we can define a simple new component. This component, however, won't handle the event; events are for component users only. Instead, the component can either handle the Windows message directly or override a method, as described in the next two sections.

Overriding Message Handlers: The Numeric Edit Box

To customize an edit box component to restrict the input it will accept, all you need to do is handle the `wm_Char` Windows messages that occur when the user presses any but a few specific keys (namely, the numeric characters).

One way to respond to a message for a given window (whether it's a form or a component) is to create a new *message-response* method that you declare using the `message` keyword. Delphi's message-handling system makes sure that your message-response method has a chance to respond to a given message before the form or component's default message handler does. You'll see in the next section that, instead of creating a new method (as we do here), you can override an existing virtual method that responds to a given message. Below is the code of the TMdNumEdit class:

```
type
  TMdNumEdit = class (TCustomEdit)
  private
    fInputError: TNotifyEvent;
  protected
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
  public
    procedure WmChar (var Msg: TWmChar); message wm_Char;
    constructor Create (Owner: TComponent); override;
  published
    property OnInputError: TNotifyEvent read fInputError write fInputError;
    property Value: Integer read GetValue write SetValue default 0;
    property AutoSelect;
    property AutoSize;
    property BorderStyle;
    // and so on...
```

This component inherits from TCustomEdit instead of TEdit so that it can hide the Text property and surface the Integer Value property instead. Notice that I don't create a new field to store this value, because we can use the existing (but now unpublished) Text property. To do this, we'll simply convert the numeric value to and from a text string. The TCustomEdit class (or actually the Windows control it wraps) automatically paints the information from the Text property on the surface of the component:

```
function TMdNumEdit.GetValue: Integer;
begin
  // set to 0 in case of error
  Result := StrToIntDef (Text, 0);
end;

procedure TMdNumEdit.SetValue (Value: Integer);
begin
  Text := IntToStr (Value);
end;
```

The most important method is the response for the wm_Char message. In the body of this method, the component filters out all the nonnumeric characters and raises a specific event in case of an error:

```
procedure TMdNumEdit.WmChar (var Msg: TWmChar);
begin
```

```
    if not (Char (Msg.CharCode) in ['0'..'9']) and not (Msg.CharCode = 8) then
    begin
      if Assigned (fInputError) then
        fInputError (Self);
    end
    else
      inherited;
  end;
```

This method checks each character as the user enters it, testing for numerals and the Backspace key (which has an ASCII value of 8). The user should be able to use Backspace in addition to the system keys (the arrow keys and Del), so we need to check for that value. We don't have to check for the system keys, because they are surfaced by a different Windows message, wm_SysChar.

That's it. Now if you place this component on a form, you can type something in the edit box and see how it behaves. You might also want to attach a method to the OnInputError event to provide feedback to the user when a wrong key is typed.

A Numeric Edit with Thousands Separators

As a further extension to the example, when typing large numbers it would be nice for the thousands separators to automatically appear and update themselves as required by the user input. You can do this by overriding the internal Change method and formatting the number properly. There are only a couple of small problems to consider. The first is that to format the number you need to have one, but the text of the edit with the thousands separators (possibly misplaced) cannot be converted to a number directly. I've written a modified version of the StringToFloat function, called StringToFloatSkipping, to accomplish this.

The second small problem is that if you modify the text of the edit box the current position of the cursor will get lost. So you need to save the original cursor position, reformat the number, and then reapply the cursor position considering that if a separator has been added or removed, it should change accordingly. All these considerations are summarized by the following complete code of the TMdThousandEdit class:

```
  type
    TMdThousandEdit = class (TMdNumEdit)
    public
      procedure Change; override;
    end;

  function StringToFloatSkipping (s: string): Extended;
  var
    s1: string;
    I: Integer;
```

```
begin
  // remove non-numbers, but keep the decimal separator
  s1 := '';
  for i := 1 to length (s) do
   if s[i] in ['0'..'9'] then
     s1 := s1 + s[i];
  Result := StrToFloat (s1);
end;

procedure TMdThousandEdit.Change;
var
  CursorPos, // original position of the cursor
  LengthDiff: Integer; // number of new separators (+ or -)
begin
  if Assigned (Parent) then
  begin
    CursorPos := SelStart;
    LengthDiff := Length (Text);
    Text := FormatFloat ('#,###',
      StringToFloatSkipping (Text));
    LengthDiff := Length (Text) - LengthDiff;
    // move the cursor to the proper position
    SelStart := CursorPos + LengthDiff;
  end;
  inherited;
end;
```

Overriding Dynamic Methods: The Sound Button

Our next component, TMdSoundButton, plays one sound when you press the button and another sound when you release it. The user specifies each sound by modifying two String properties that name the appropriate WAV files for the respective sounds. Once again, we need to intercept and modify some system messages (wm_LButtonDown and wm_LButtonUp), but instead of handling the messages by writing a new message-response method, we'll override the appropriate *second-level* handlers.

NOTE When most VCL components handle a Windows message, they call a *second-level* message handler (usually a dynamic method), instead of executing code directly in the message-response method. This makes it simpler for you to customize the component in a derived class. Typically, a second-level handler will do its own work and then call any event handler that the component user has assigned.

Here is the code of the TMdSoundButton class, with the two protected methods that override the second-level handlers, and the two string properties that identify the sound files.

You'll notice that in the property declarations, we read and write the corresponding private fields without calling a get or set method, simply because we don't need to do anything special when the user makes changes to those properties.

```
type
  TMdSoundButton = class(TButton)
  private
    FSoundUp, FSoundDown: string;
  protected
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  published
    property SoundUp: string read FSoundUp write FSoundUp;
    property SoundDown: string read FSoundDown write FSoundDown;
  end;
```

There are several reasons why overriding existing second-level handlers is generally a better approach than handling straight Windows messages. First, this technique is more sound from an object-oriented perspective. Instead of duplicating the message-response code from the base class and then customizing it, you're overriding a virtual method call that the VCL designers planned for you to override. Second, if someone needs to derive another class from one of your component classes, you'll want to make it as easy for them to customize as possible, and overriding second-level handlers is less likely to induce strange errors (if only because you're writing less code). Finally, this will make your component classes more consistent with VCL—and therefore easier for someone else to figure out. Here is the code of the two second-level handlers:

```
uses
  MMSystem;

procedure TMdSoundButton.MouseDown(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
  PlaySound (PChar (FSoundDown), 0, snd_Async);
end;

procedure TMdSoundButton.MouseUp(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseUp (Button, Shift, X, Y);
  PlaySound (PChar (FSoundUp), 0, snd_Async);
end;
```

In both cases, you'll notice that we call the inherited version of the methods *before* we do anything else. For most second-level handlers, this is a good practice, since it ensures that we execute the standard behavior before we execute any custom behavior.

Next, you'll notice that we call the PlaySound Win32 API function to play the sound. You can use this function (which is defined in the MmSystem unit to play either WAV files or system sounds, as the SoundB example demonstrates. Here is a textual description of the form of this sample program (from the DFM file):

```
object MdSoundButton1: TMdSoundButton
  Caption = 'Press'
  SoundUp = 'RestoreUp'
  SoundDown = 'RestoreDown'
end
```

NOTE Selecting a proper value for these sound properties is far from simple. Later in this chapter, I'll show you how to add a property editor to the component to simplify the operation.

Handling Internal Messages: The Active Button

The Windows interface is evolving toward a new standard, including components that become highlighted as the mouse cursor moves over them. Delphi provides similar support in many of its built-in components, but what does it take to mimic this behavior for a simple button? This might seem a complex task to accomplish, but it is not.

The development of a component can become much simpler once you know which virtual function to override or which message to hook onto. The next component, the TMdActiveButton class, demonstrates this by handling some internal Delphi messages to accomplish its task in a very simple way. (For information about where these internal Delphi messages come from, see the sidebar "Component Messages and Notifications.")

The ActiveButton component handles the cm_MouseEnter and cm_MouseExit internal Delphi messages, which are received when the mouse cursor enters or leaves the area corresponding to the component:

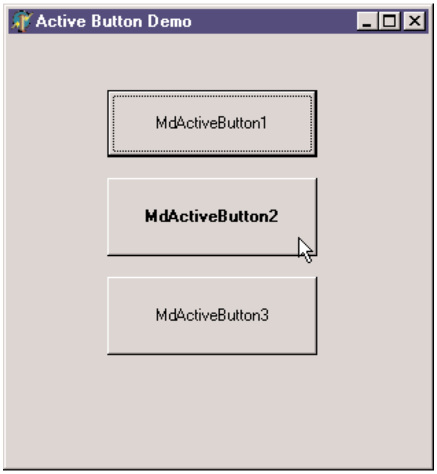
```
type
  TMdActiveButton = class (TButton)
  protected
    procedure MouseEnter (var Msg: TMessage);
      message cm_mouseEnter;
    procedure MouseLeave (var Msg: TMessage);
      message cm_mouseLeave;
  end;
```

The code you write for these two methods can do whatever you want. For this example, I've decided to simply toggle the bold style of the font of the button itself. You can see the effect of moving the mouse over one of these components in Figure 11.10.

```
procedure TMdActiveButton.MouseEnter (var Msg: TMessage);
begin
  Font.Style := Font.Style + [fsBold];
end;

procedure TMdActiveButton.MouseLeave (var Msg: TMessage);
begin
  Font.Style := Font.Style - [fsBold];
end;
```

FIGURE 11.10:

An example of the use of the ActiveButton component



You can add other effects at will, including enlarging the font itself, making the button the default, or increasing its size a little. The best effects usually involve colors, but you should inherit from the TBitBtn class to have this support (TButton controls have a fixed color).

Component Messages and Notifications

To build the ActiveButton component, I've used two internal Delphi component messages, as indicated by their *cm* prefix. These messages can be quite interesting, as the example highlights, but they are almost completely undocumented by Borland. There is also a second group of internal Delphi messages, indicated as component notifications and distinguished by their *cn* prefix. I don't have enough space here to discuss each of them or provide a detailed analysis; browse the VCL source code if you want to learn more.

WARNING As this is a rather advanced topic, feel free to skip this section if you are new to writing Delphi components. But component messages are not documented in the Delphi help file, so I felt it was important to at least *list* them here.

Component Messages

A Delphi component passes *component messages* to other components to indicate any change in its state that might affect those components. Most of these messages start as Windows messages, but some of them are more complex, higher-level translations and not simple remappings. Also, components send their own messages as well as forwarding those received from Windows. For example, changing a property value or some other characteristic of the component may necessitate telling one or more other components about the change.

We can group these messages into categories:

- Activation and input focus messages are sent to the component being activated or deactivated, receiving or losing the input focus:

`cm_Activate`	Corresponds to the `OnActivate` event of forms and of the application
`cm_Deactivate`	Corresponds to `OnDeactivate`
`cm_Enter`	Corresponds to `OnEnter`
`cm_Exit`	Corresponds to `OnExit`
`cm_FocusChanged`	Sent whenever the focus changes between components of the same form (later, we'll see an example using this message)
`cm_GotFocus`	Declared but not used
`cm_LostFocus`	Declared but not used

- Messages sent to child components when a property changes:

`cm_BiDiModeChanged`

`cm_BorderChanged`

`cm_ColorChanged`

`cm_Ctl3DChanged`

`cm_CursorChanged`

`cm_EnabledChanged`

`cm_FontChanged`

`cm_IconChanged`

`cm_ShowHintChanged`

`cm_ShowingChanged`

`cm_SysFontChanged`

`cm_TabStopChanged`

`cm_TextChanged`

`cm_VisibleChanged`

Monitoring these messages can help track changes in a property. You might need to respond to these messages in a new component, but it's not likely.

- Messages related to *ParentXxx* properties: cm_ParentFontChanged, cm_ParentColorChanged, cm_ParentCtl3DChanged, cm_ParentBiDiModeChanged, and cm_ParentShowHintChanged. These are very similar to the messages of the previous group.
- Notifications of changes in the Windows system: cm_SysColorChange, cm_WinIniChange, cm_TimeChange, and cm_FontChange. Handling these messages is useful only in special components that need to keep track of system colors or fonts.
- Mouse messages: cm_Drag is sent many times during dragging operations. cm_MouseEnter and cm_MouseLeave are sent to the control when the cursor enters or leaves its surface, but these are sent by the Application object as low-priority messages. cm_MouseWheel corresponds to wheel-based operations.

  cm_Drag has a DragMessage parameter that indicates a sort of submessage, and the address of the TDragRec record that indicates the mouse position and the components involved in the dragging operation. The cm_Drag message isn't that important, because Delphi defines many drag events and drag methods you can override. However, you can respond to cm_Drag for a few things that don't generate an event or method call. This message is sent to find the target component (when the DragMessage field is dmFindTarget); to indicate that the cursor has reached a component (the dmDragEnter submessage), is being moved over it (dmDragMove), or has left it (dmDragLeave); when the drop operation is accepted (dmDragDrop); and when it is aborted (dmDragCancel).
- Application messages:

cm_AppKeyDown	Sent to the Application object to let it determine whether a key corresponds to a menu shortcut
cm_AppSysCommand	Corresponds to the wm_SysCommand message
cm_DialogHandle	Sent in a DLL to retrieve the value of the DialogHandle property (used by some dialog boxes not built with Delphi)
cm_InvokeHelp	Sent by code in a DLL to call the InvokeHelp method
cm_WindowHook	Sent in a DLL to call the HookMainWindow and UnhookMainWindow methods

You'll rarely need to use these messages yourself. There is also a cm_HintShowPause message, which is apparently never handled in VCL.

- Delphi internal messages:
  - `cm_CancelMode` Terminates special operations, such as showing the pull-down list of a combo box
  - `cm_ControlChange` Sent to each control before adding or removing a child control (handled by some common controls)
  - `cm_ControlListChange` Sent to each control before adding or removing a child control (handled by the DBCtrlGrid component)
  - `cm_DesignHitTest` Determines whether a mouse operation should go to the component or to the form designer
  - `cm_HintShow` Sent to a control just before displaying its hint (only if the `ShowHint` property is True)
  - `cm_HitTest` Sent to a control when a parent control is trying to locate a child control at a given mouse position (if any)
  - `cm_MenuChanged` Sent after MDI or OLE menu-merging operations
- Messages related to special keys:
  - `cm_ChildKey` Sent to the parent control to handle some special keys (in Delphi, this message is handled only by DBCtrlGrid components)
  - `cm_DialogChar` Sent to a control to determine whether a given input key is its accelerator character
  - `cm_DialogKey` Handled by modal forms and controls that need to perform special actions
  - `cm_IsShortCut` I haven't yet figured out the exact role of this new message.
  - `cm_WantSpecialKey` Handled by controls that interpret special keys in an unusual way (for example, using the Tab key for navigation, as some Grid components do)

- Messages for specific components:

`cm_GetDataLink`	Used by DBCtrlGrid controls (and discussed in Chapter 18)
`cm_TabFontChanged`	Used by the TabbedNotebook components
`cm_ButtonPressed`	Used by SpeedButtons to notify other sibling SpeedButton components (to enforce radio-button behavior)
`cm_DeferLayout`	Used by DBGrid components

- OLE container messages: `cm_DocWindowActivate`, `cm_IsToolControl`, `cm_Release`, `cm_UIActivate`, and `cm_UIDeactivate`.
- Dock-related messages, including `cm_DockClient`, `cm_DockNotification`, `cmFloat`, and `cm_UndockClient`.
- Method-implementation messages, such as `cm_RecreateWnd`, called inside the `RecreateWnd` method of `TControl`; `cm_Invalidate`, called inside `TControl.Invalidate`; `cm_Changed`, called inside `TControl.Changed`; and `cm_AllChildrenFlipped`, called in the `DoFlipChildren` methods of `TWinControl` and `TScrollingWinControl`. In the similar group fall two action list–related messages, `cm_ActionUpdate` and `cm_ActionExecute`.

Finally, there are messages defined and handled by specific components and declared in the respective units, such as `cm_DeferLayout` for DBGrid controls and a group of almost 10 messages for action bar components.

Component Notifications

Component notification messages are those sent from a parent form or component to its children. These notifications correspond to messages sent by Windows to the parent control's window, but logically intended for the control. For example, interaction with controls such as buttons, edit, or list boxes, causes Windows to send a `wm_Command` message to the parent of the control. When a Delphi program receives these messages, it forwards the message to the control itself, as a notification. The Delphi control can handle the message and eventually fire an event. Similar dispatching operations take place for many other commands.

The connection between Windows messages and component notification ones is so tight that you'll often recognize the name of the Windows message from the name of the notification message, simply replacing the initial *cn* with *wm*. There are several distinct groups of component notification messages:

- General keyboard messages: `cn_Char`, `cn_KeyUp`, `cn_KeyDown`, `cn_SysChar`, and `cn_SysKeyDown`.

- Special keyboard messages used only by list boxes with the lbs_WantKeyboardInput style: cn_CharToItem and cn_VKeyToItem.
- Messages related to the owner-draw technique: cn_CompareItem, cn_DeleteItem, cn_DrawItem, and cn_MeasureItem.
- Messages for scrolling, used only by scroll bar and track bar controls: cn_HScroll and cn_VScroll.
- General notification messages, used by most controls: cn_Command, cn_Notify, and cn_ParentNotify.
- Control color messages: cn_CtlColorBtn, cn_CtlColorDlg, cn_CtlColorEdit, cn_CtlColorListbox, cn_CtlColorMsgbox, cn_CtlColorScrollbar, and cn_CtlColorStatic.

Some more control notifications are defined for common controls support (in the ComCtrls unit).

An Example of Component Messages

As a very simple example of the use of some component messages, I've written the CMNTest program. This program has a form with three edit boxes, and associated labels. The first message it handles, cm_DialogKey, allows it to treat the Enter key as if it were a Tab key. The code of this method checks for the Enter key's code and sends the same message, but passes the vk_Tab key code. To halt further processing of the Enter key, we set the result of the message to 1:

```
procedure TForm1.CMDialogKey(var Message: TCMDialogKey);
begin
  if (Message.CharCode = VK_RETURN) then
  begin
    Perform (CM_DialogKey, VK_TAB, 0);
    Message.Result := 1;
  end
  else
    inherited;
end;
```

The second message, cm_DialogChar, monitors accelerator keys. This can be useful to provide custom shortcuts without defining an extra menu for them. In this case, I'm simply logging the special keys in a label:

```
procedure TForm1.CMDialogChar(var Msg: TCMDialogChar);
begin
  Label1.Caption := Label1.Caption + Char (Msg.CharCode);
  inherited;
end;
```

Finally, the form handles the cm_FocusChanged message, to respond to focus changes without having to handle the OnEnter event of each of its components. Again, the simple action is to display a description of the focused component:

```
procedure TForm1.CmFocusChanged(var Msg: TCmFocusChanged);
begin
  Label5.Caption := 'Focus on ' + Msg.Sender.Name;
end;
```

The advantage of this approach is that it works independently of the type and number of components you add to the form, and it does so without any special action on your part. Again, this is a trivial example for such an advanced topic, but if you add to this the code of the ActiveButton component, you have at least a few reasons to look into these special, undocumented messages. At times, writing the same code without their support can become extremely complex.

A Nonvisual Dialog Component

The next component we'll examine is completely different from the ones we have seen up to now. After building window-based controls and simple graphic components, I'm now going to build a nonvisual component.

The basic idea is that forms are components. When you have built a form that might be particularly useful in multiple projects, you can add it to the Object Repository or make a component out of it. The second approach is more complex than the first, but it makes using the new form easier and allows you to distribute the form without its source code. As an example, I'll build a component based on a custom dialog box, trying to mimic as much as possible the behavior of standard Delphi dialog box components.

The first step in building a dialog box in a component is to write the code of the dialog box itself, using the standard Delphi approach. Just define a new form and work on it as usual. When a component is based on a form, you can almost visually design the component. Of course, once the dialog box has been built, you have to define a component around it in a nonvisual way.

The standard dialog box I want to build is based on a list box, because it is common to let a user choose a value from a list of strings. I've customized this common behavior in a dialog box and then used it to build a component. The simple ListBoxForm form I've built has a list box and the typical OK and Cancel buttons, as shown in its textual description:

```
object MdListBoxForm: TMdListBoxForm
  BorderStyle = bsDialog
  Caption = 'ListBoxForm'
  object ListBox1: TListBox
```

```
    OnDblClick = ListBox1DblClick
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

The only method of this dialog box form relates to the double-click event of the list box, which closes the dialog box as though the user clicked the OK button, by setting the ModalResult property of the form to mrOk. Once the form works, we can start changing its source code, adding the definition of a component and removing the declaration of the global variable for the form.

NOTE For components based on a form, you can use two Pascal source code files: one for the form and the other for the component encapsulating it. It is also possible to place both the component and the form in a single unit, as I've done for this example. In theory it would be even nicer to declare the form class in the implementation portion of this unit, hiding it from the users of the component. In practice this is not a good idea. To manipulate the form visually in the Form Designer, the form class declaration must appear in the interface section of the unit. The rationale behind this behavior of the Delphi IDE is that, among other things, this constraint minimizes the amount of code the module manager has to scan to find the form declaration—an operation that must be performed often to maintain the synchronization of the visual form with the form class definition.

The most important of these operations is the definition of the TMdListBoxDialog component. This component is defined as "nonvisual" because its immediate ancestor class is TComponent. The component has one public property and these three published properties:

- Lines is a TStrings object, which is accessed via two methods, GetLines and SetLines. This second method uses the Assign procedure to copy the new values to the private field corresponding to this property. This internal object is initialized in the Create constructor and destroyed in the Destroy method.
- Selected is an integer that directly accesses the corresponding private field. It stores the selected element of the list of strings.
- Title is a string used to change the title of the dialog box.

The public property is SelItem, a read-only property that automatically retrieves the selected element of the list of strings. Notice that this property has no storage and no data: it simply accesses other properties, providing a virtual representation of data:

```
type
  TMdListBoxDialog = class (TComponent)
  private
    FLines: TStrings;
    FSelected: Integer;
    FTitle: string;
    function GetSelItem: string;
    procedure SetLines (Value: TStrings);
    function GetLines: TStrings;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Execute: Boolean;
    property SelItem: string read GetSelItem;
  published
    property Lines: TStrings read GetLines write SetLines;
    property Selected: Integer read FSelected write FSelected;
    property Title: string read FTitle write FTitle;
  end;
```

Most of the code of this example is in the Execute method, a function that returns True or False depending on the modal result of the dialog box. This is consistent with the Execute method of most standard Delphi dialog box components. The Execute function creates the form dynamically, sets some of its values using the component's properties, shows the dialog box, and if the result is correct, updates the current selection:

```
function TMdListBoxDialog.Execute: Boolean;
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create ('No items in the list');
  ListBoxForm := TListBoxForm.Create (Self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
    begin
      Result := True;
      Selected := ListBoxForm.ListBox1.ItemIndex;
    end
    else
      Result := False;
  finally
    ListBoxForm.Free;
  end;
end;
```

Notice that the code is contained within a try/finally block, so if a run-time error occurs when the dialog box is displayed, the form will be destroyed anyway. I've also used exceptions to raise an error if the list is empty when a user runs it. This error is by design, and using an exception is a good technique to enforce it. The other methods of the component are quite straightforward. The constructor creates the FLines string list, which is deleted by the destructor; the GetLines and SetLines methods operate on the string list as a whole; and the GetSelItem function (listed below) returns the text of the selected item:

```
function TMdListBoxDialog.GetSelItem: string;
begin
  if (Selected >= 0) and (Selected < FLines.Count) then
    Result := FLines [Selected]
  else
    Result := '';
end;
```

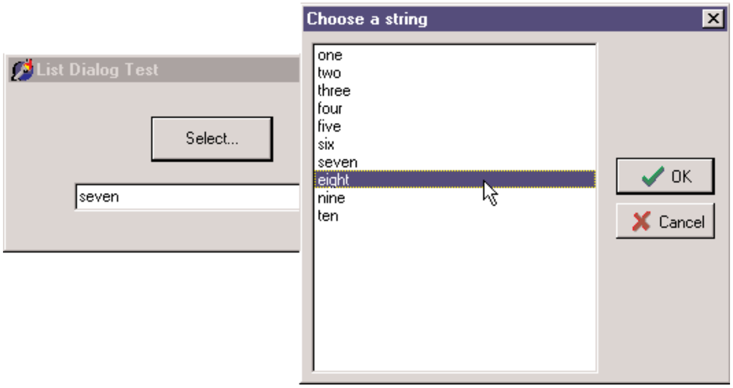
Of course, since we are manually writing the code of the component and adding it to the source code of the original form, we have to remember to write the Register procedure.

Using the Nonvisual Component

Once you've done that and the component is ready, you must provide a bitmap. For nonvisual components, bitmaps are very important because they are used not only for the Component Palette, but also when you place the component on a form. After preparing the bitmap and installing the component, I've written a simple project to test it. The form of this test program has a button, an edit box, and the MdListDialog component. In the program, I've added only a few lines of code, corresponding to the OnClick event of the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // select the text of the edit, if corresponding to one of the strings
  MdListDialog1.Selected := MdListDialog1.Lines.IndexOf (Edit1.Text);
  // run the dialog and get the result
  if MdListDialog1.Execute then
    Edit1.Text := MdListDialog1.SelItem;
end;
```

That's all you need to run the dialog box we have placed in the component, as you can see in Figure 11.11. As you've seen, this is an interesting approach to the development of some common dialog boxes.



List Dialog Test

Select...

seven

Choose a string

one
two
three
four
five
six
seven
eight
nine
ten

FIGURE 11.11:

The ListDialDemo example shows the dialog box I've encapsulated in the ListDial component.

Defining Custom Actions

Besides defining custom components, you can define and register new standard actions, which will be made available in the Action Editor of the Action List component. Creating new actions is not complex. You have to inherit from the TAction class and override some of the methods of the base class.

There are basically three methods to override. The HandlesTarget function returns whether the action object wants to handle the operation for the current target, which is by default the control with the focus. The UpdateTarget procedure can set the user interface of the controls connected with the action, eventually disabling the action if the operation is currently not available. Finally, you can implement the ExecuteTarget method to determine the actual code to execute, so that the user can simply select the action and doesn't have to implement it.

To show you this approach in practice, I've implemented the three cut, copy, and paste actions for a list box, in a way similar to what VCL does for an edit box (although I've actually simplified the code a little). I've written a base class, which inherits from the generic TListControlAction class of the new ExtActns unit. This base class, TMdCustomListAction, adds some common code, shared by all the specific actions, and publishes a few action properties. The three derived classes have their own ExecuteTarget code, plus little more. Here are the four classes:

```
type
  TMdCustomListAction = class (TListControlAction)
  protected
    function TargetList (Target: TObject): TCustomListBox;
    function GetControl (Target: TObject): TCustomListControl;
  public
```

```
    procedure UpdateTarget (Target: TObject); override;
  published
    property Caption;
    property Enabled;
    property HelpContext;
    property Hint;
    property ImageIndex;
    property ListControl;
    property ShortCut;
    property SecondaryShortCuts;
    property Visible;
    property OnHint;
  end;

  TMdListCutAction = class (TMdCustomListAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;

  TMdListCopyAction = class (TMdCustomListAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;

  TMdListPasteAction = class (TMdCustomListAction)
  public
    procedure UpdateTarget (Target: TObject); override;
    procedure ExecuteTarget (Target: TObject); override;
  end;
```

The HandlesTarget method, one of the three key methods of actions, is provided by the TListControlAction class, with this code:

```
function TListControlAction.HandlesTarget(Target: TObject): Boolean;
begin
  Result := ((ListControl <> nil) or
    (ListControl = nil) and (Target is TCustomListControl)) and
    TCustomListControl(Target).Focused;
end;
```

The UpdateTarget method, instead, has two different implementations. The default one is provided by the base class and used by the copy and cut actions. These actions are enabled only if the target list box has at least one item and an item is currently selected. The status of the paste action depends instead on the Clipboard status:

```
procedure TMdCustomListAction.UpdateTarget (Target: TObject);
begin
  Enabled := (TargetList (Target).Items.Count > 0)
```

```
    and (TargetList (Target).ItemIndex >= 0);
end;

function TMdCustomListAction.TargetList (Target: TObject): TCustomListBox;
begin
  Result := GetControl (Target) as TCustomListBox;
end;

function TMdCustomListAction.GetControl(Target: TObject): TCustomListControl;
begin
  Result := Target as TCustomListControl;
end;

procedure TMdListPasteAction.UpdateTarget (Target: TObject);
begin
  Enabled := Clipboard.HasFormat (CF_TEXT);
end;
```

The TargetList function uses the GetControl function of the TListControlAction class, which returns either the list box connected to the action at design time or the target control, the list box control with the input focus.

Finally, the three ExecuteTarget methods simply perform the corresponding actions on the target list box:

```
procedure TMdListCopyAction.ExecuteTarget (Target: TObject);
begin
  with TargetList (Target) do
    Clipboard.AsText := Items [ItemIndex];
end;

procedure TMdListCutAction.ExecuteTarget(Target: TObject);
begin
  with TargetList (Target) do
  begin
    Clipboard.AsText := Items [ItemIndex];
    Items.Delete (ItemIndex);
  end;
end;

procedure TMdListPasteAction.ExecuteTarget(Target: TObject);
begin
  (TargetList (Target)).Items.Add (Clipboard.AsText);
end;
```

Once you've written this code in a unit and added it to a package (in this case, the MdPack package), the final step is to register the new custom actions in a given category. This is indicated as the first parameter of the `RegisterActions` procedure, while the second is the list of action classes to register:

```
procedure Register;
begin
  RegisterActions ('List',
    [TMdListCutAction, TMdListCopyAction, TMdListPasteAction], nil);
end;
```

To test the use of these three custom actions, I've written the ListTest example on the companion CD. This program has two list boxes plus a toolbar that contains three buttons connected with the three custom actions and an edit box for entering new values. The program allows a user to cut, copy, and paste list box items. Nothing special, you might think, but the strange fact is that the program has absolutely no code!

Writing Property Editors

Writing components is certainly an effective way to customize Delphi, helping developers to build applications faster without requiring a detailed knowledge of low-level techniques. The Delphi environment is also quite open to extensions. In particular, you can extend the Object Inspector by writing custom property editors and to extend the Form Designer by adding component editors.

NOTE Along with these techniques, Delphi offers some internal interfaces to add-on tool developers. Using these interfaces, known as OpenTools API, requires an advanced understanding of how the Delphi environment works and a fairly good knowledge of many advanced techniques that are not discussed in this book. You can find technical information and some examples of these techniques on my Web site, `www.marcocantu.com`, along with links to other sites where these techniques are presented.

Every property editor must be a subclass of the abstract `TPropertyEditor` class, which is defined in the DesignEditors unit of the ToolsApi and provides a standard implementation for the `IProperty` interface.

NOTE The Tools API in Delphi 6 has changed considerably, also for consistency with Kylix. For example, the DsgnIntf unit of Delphi 5 has been split into the units DesignIntf, DesignEditors, and other specific units. Borland has also introduced interfaces to define the sets of methods of each kind of editor. However, most of the simpler examples, such as those presented in this book, compile almost unchanged. As this is not an in-depth analysis of the Tools API, I'm not providing a list of changes from Delphi 5 to Delphi 6, although they are substantial. For more information, you can study the extensive source code in the `\Source\ToolsApi` directory of Delphi 6.

Delphi already defines some specific property editors for strings (the TStringProperty class), integers (the TIntegerProperty class), characters (the TCharProperty class), enumerations (the TEnumProperty class), sets (the TSetProperty class), so you can actually inherit your property editor from the one of the type of property you are working with.

In any custom property editor, you have to redefine the GetAttributes function so it returns a set of values indicating the capabilities of the editor. The most important attributes are paValueList and paDialog. The paValueList attribute indicates that the Object Inspector will show a combo box with a list of values (eventually sorted if the paSortList attribute is set) provided by overriding the GetValues method. The paDialog attribute style activates an ellipsis button in the Object Inspector, which executes the Edit method of the editor.

An Editor for the Sound Properties

The sound button we built earlier had two sound-related properties: SoundUp and SoundDown. These were actually strings, so we were able to display them in the Object Inspector using a default property editor. However, requiring the user to type the name of a system sound or an external file is not very friendly, and it's a bit error-prone.

When you need to select a file for a string property, you can reuse an existing property editor, the TMPFilenameProperty class. All you have to do is register this editor for the property using the special RegisterPropertyEditor procedure, as in:

```
RegisterPropertyEditor (TypeInfo (string), TDdhSoundButton, 'SoundUp',
  TMPFileNameProperty);
```

This editor allows you to select a file for the sound, but we want to be able to choose the name of a system sound as well. (As described earlier, system sounds are predefined names of sounds connected with user operations, associated with actual sound files in the Sounds applet of the Windows Control Panel.) For this reason, instead of using this simple approach I'll build a more complex property editor. My editor for sound strings allows a user to either choose a value from a drop-down list or display a dialog box from which to load and test a sound (from a sound file or a system sound). For this reason, the property editor provides both Edit and GetValues methods:

```
type
  TSoundProperty = class (TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure Edit; override;
  end;
```

TIP The default Delphi convention is to name a property editor class with a name ending with *Property* and all component editors with a name ending with *Editor*.

The GetAttributes function combines both the paValueList (for the drop-down list) and the paDialog attributes (for the custom edit box), and also sorts the lists and allows the selection of the property for multiple components:

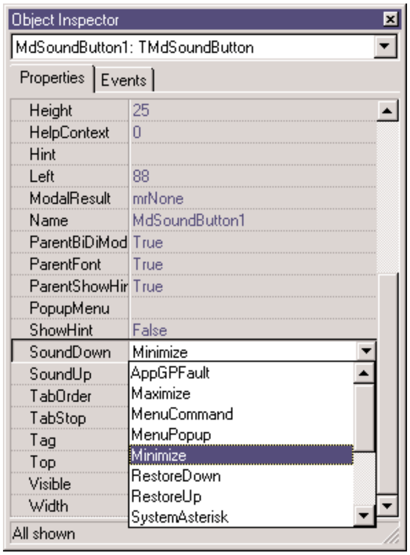
```
function TSoundProperty.GetAttributes: TPropertyAttributes;
begin
  // editor, sorted list, multiple selection
  Result := [paDialog, paMultiSelect, paValueList, paSortList];
end;
```

The GetValues method simply calls the procedure it receives as parameter many times, once for each string it wants to add to the drop-down list (as you can see in Figure 11.12):

```
procedure TSoundProperty.GetValues(Proc: TGetStrProc);
begin
  // provide a list of system sounds
  Proc ('Maximize');
  Proc ('Minimize');
  Proc ('MenuCommand');
  Proc ('MenuPopup');
  Proc ('RestoreDown');
  Proc ('RestoreUp');
  Proc ('SystemAsterisk');
  Proc ('SystemDefault');
  Proc ('SystemExclamation');
  Proc ('SystemExit');
  Proc ('SystemHand');
  Proc ('SystemQuestion');
  Proc ('SystemStart');
  Proc ('AppGPFault');
end;
```

FIGURE 11.12:

The list of sounds provides a hint for the user, who can also type in the property value or double-click to activate the editor (shown later, in Figure 11.13).



Height	25
HelpContext	0
Hint	
Left	88
ModalResult	mrNone
Name	MdSoundButton1
ParentBiDiMod	True
ParentFont	True
ParentShowHir	True
PopupMenu	
ShowHint	False
SoundDown	Minimize
SoundUp	
TabOrder	
TabStop	
Tag	
Top	
Visible	
Width	

A better approach would be to extract these values from the Windows Registry, where all these names are listed. The Edit method is very straightforward, as it simply creates and displays a dialog box. You'll notice that we could have just displayed the Open dialog box directly, but we decided to add an intermediate step to allow the user to test the sound. This is similar to what Delphi does with graphic properties. You open the preview first, and load the file only after you've confirmed that it's correct. The most important step is to load the file and test it before you apply it to the property. Here is the code of the Edit method:

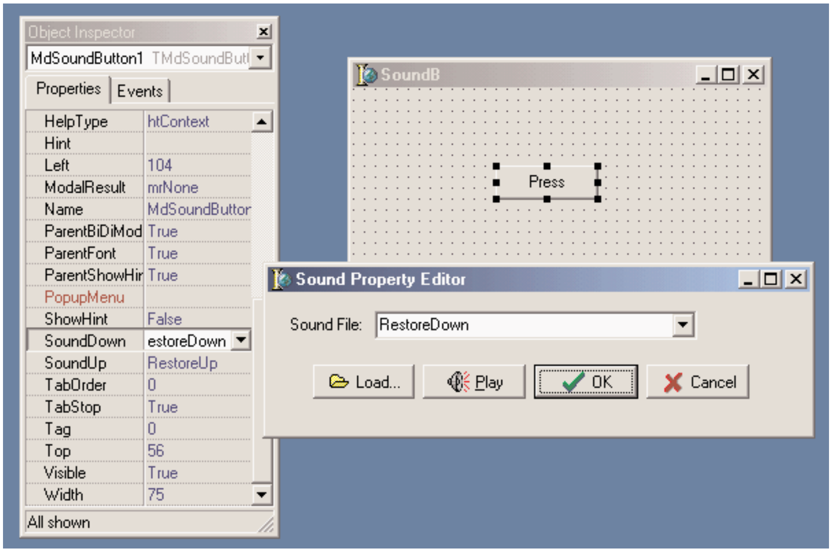
```
procedure TSoundProperty.Edit;
begin
  SoundForm := TSoundForm.Create (Application);
  try
    SoundForm.ComboBox1.Text := GetValue;
    // show the dialog box
    if SoundForm.ShowModal = mrOK then
      SetValue (SoundForm.ComboBox1.Text);
  finally
    SoundForm.Free;
  end;
end;
```

The GetValue and SetValue methods called above are defined by the base class, the string property editor. They simply read and write the value of the current component's property that we are editing. As an alternative, you can access the component you're editing by using the GetComponent method (which requires a parameter indicating which of the selected components you are working on—0 indicates the first component). When you access the component directly, you also need to call the Modified method of the Designer object (a property of the base class property editor). We don't need this Modified call in the example, as the base class SetValue method does this automatically for us.

The Edit method above displays a dialog box, a standard Delphi form that is built visually, as always, and added to the package hosting the design-time components. The form is quite simple; a ComboBox displays the values returned by the GetValues method, and four buttons allow you to open a file, test the sound, and terminate the dialog box by accepting the values or canceling. You can see an example of the dialog box in Figure 11.13. Providing a drop-down list of values *and* a dialog box for editing a property causes the Object Inspector to display only the arrow button that indicates a drop-down list and to omit the ellipsis button to indicate that a dialog box editor is available.

FIGURE 11.13:

The Sound Property Editor's form displays a list of available sounds and lets you load a file and hear the selected sound.



HelpType	htContext
Hint	
Left	104
ModalResult	mrNone
Name	MdSoundButtor
ParentBiDiMod	True
ParentFont	True
ParentShowHir	True
PopupMenu	
ShowHint	False
SoundDown	estoreDown
SoundUp	RestoreUp
TabOrder	0
TabStop	True
Tag	0
Top	56
Visible	True
Width	75

All shown

Sound File: RestoreDown

The first two buttons of the form each have a simple method assigned to their `OnClick` event:

```
procedure TSoundForm.btnLoadClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    ComboBox1.Text := OpenDialog1.FileName;
end;

procedure TSoundForm.btnPlayClick(Sender: TObject);
begin
  PlaySound (PChar (ComboBox1.Text), 0, snd_Async);
end;
```

Unfortunately, I haven't found a simple way to determine whether a sound is properly defined and is available. (Checking the file is possible, but the system sounds create a few issues.) The `PlaySound` function returns an error code when played synchronously, but only if it can't find the default system sound it attempts to play if it can't find the sound you asked for. If the requested sound is not available, it plays the default system sound and doesn't return the error code. `PlaySound` looks for the sound in the Registry first and, if it doesn't find the sound there, checks to see whether the specified sound file exists.

TIP

If you want to further extend this example, you might add graphics to the drop-down list displayed in the Object Inspector—if you can decide which graphics to attach to particular sounds.

Installing the Property Editor

After you've written this code, you can install the component and its property editor in Delphi. To accomplish this, you have to add the following statement to the Register procedure of the unit:

```
procedure Register;
begin
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundUp',
    TSoundProperty);
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundDown',
    TSoundProperty);
end;
```

This call registers the editor specified in the last parameter for use with properties of type string (the first parameter), but only for a specific component and for a property with a specific name. These last two values can be omitted to provide more general editors. Registering this editor allows the Object Inspector to show a list of values and the dialog box called by the Edit method.

To install this component we can simply add its source code file into an existing or new package. Instead of adding this unit and the others of this chapter to the MdPack package, I built a second package, containing all the add-ins built in this chapter. The package is named MdDesPk (which stands for "*Mastering Delphi* design package"). What's new about this package is that I've compiled it using the {$DESIGNONLY} compiler directive. This directive is used to mark packages that interact with the Delphi environment, installing components and editors, but are not required at run time by applications you've built.

NOTE The source code of all of the add-on tools is in the MdDesPk subdirectory, along with the code of the package used to install them. There are no examples demonstrating how to use these design-time tools, because all you have to do is select the corresponding components in the Delphi environment and see how they behave.

The property editor's unit uses the SoundB unit, which defines the TMdSoundButton component. For this reason the new package should refer to the existing package. Here is its initial code (I'll add other units to it later in this chapter):

```
package MdDesPk;

{$R *.RES}
{$ALIGN ON}
...
{$DESCRIPTION 'Mastering Delphi DesignTime Package'}
{$DESIGNONLY}
```

```
requires
  vcl,
  Mdpack;

contains
  PeSound in 'PeSound.pas',
  PeFSound in 'PeFSound.pas' {SoundForm};
```

Writing a Component Editor

Using property editors allows the developer to make a component more user-friendly. In fact, the Object Inspector represents one of the key pieces of the user interface of the Delphi environment, and Delphi developers use it quite often. However, there is a second approach you can adopt to customize how a component interacts with Delphi: write a custom component editor.

Just as property editors extend the Object Inspector, component editors extend the Form Designer. In fact, when you right-click within a form at design time, you see some default menu items, plus the items added by the component editor of the selected component. Examples of these menu items are those used to activate the Menu Designer, the Fields Editor, the Visual Query Builder, and other editors of the environment. At times, displaying these special editors becomes the default action of a component when it is double-clicked.

Common uses of component editors include adding an About box with information about the developer of the component, adding the component name, and providing specific wizards to set up its properties.

Subclassing the *TComponentEditor* Class

A component editor should generally inherit from the `TComponentEditor` class, which provides the base implementation of the `IComponentEditor` interface. The most important methods of this interface are:

- `GetVerbCount` returns the number of menu items to add to the local menu of the Form Designer when the component is selected.
- `GetVerb` is called once for each new menu item and should return the text that will go in the local menu for each.
- `ExecuteVerb` is called when one of the new menu items is selected. The number of the item is passed as the method's parameter.
- `Edit` is called when the user double-clicks the component in the Form Designer to activate the default action.

Once you get used to the idea that a "verb" is nothing but a new menu item with a corresponding action to execute, the names of the methods of this interface become quite intuitive. This interface is actually much simpler than those of property editors we've seen before.

NOTE Like property editors, component editors were modified extensively from Delphi 5 to Delphi 6, and are now defined in the DesignEditors and DesignIntf units. But, again, the simpler examples like this one keep compiling almost unchanged, so I won't delve into the differences.

A Component Editor for the ListDialog

Now that I've introduced the key ideas about writing component editors, we can look at an example, an editor for the ListDialog component built earlier. In my component editor, I simply want to be able to show an About box, add a copyright notice to the menu (an improper but very common use of component editors), and allow users to perform a special action—previewing the dialog box connected with the dialog component. I also want to change the default action to simply show the About box after a beep (which is not particularly useful but demonstrates the technique).

To implement this property editor, the program must override the four methods listed above:

```
uses
  DesignIntf;

type
  TMdListCompEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
  end;
```

The first method simply returns the number of menu items I want to add to the local menu:

```
function TMdListCompEditor.GetVerbCount: Integer;
begin
  Result := 3;
end;
```

This method is called only once, before displaying the menu. The second method, instead, is called once for each menu item, so in this case it is called three times:

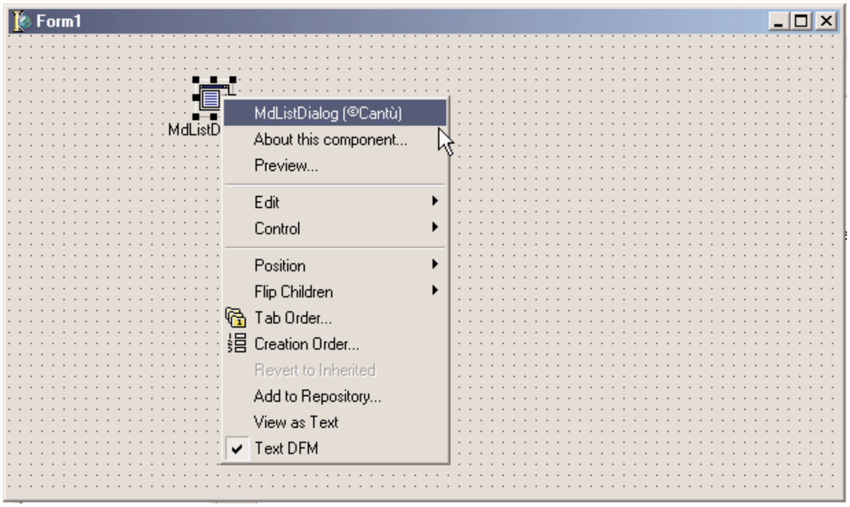
```
function TMdListCompEditor.GetVerb (Index: Integer): string;
begin
```

```
  case Index of
    0: Result := ' MdListDialog (©Cantù)';
    1: Result := '&About this component...';
    2: Result := '&Preview...';
  end;
end;
```

The effect of this code is to add the menu items to the local menu of the form, as you can see in Figure 11.14. Selecting any of these menu items just activates the ExecuteVerb method of the component editor:

```
procedure TMdListCompEditor.ExecuteVerb (Index: Integer);
begin
  case Index of
    0..1: MessageDlg ('This is a simple component editor'#13 +
      'built by Marco Cantù'#13 +
      'for the book "Mastering Delphi"', mtInformation, [mbOK], 0);
    2: with Component as TMdListDialog do
      Execute;
  end;
end;
```

FIGURE 11.14: The custom menu items added by the property editor of the ListDialog component



I decided to handle the first two items in a single branch of the case statement, although I could have skipped the code for the copyright notice item. The other command changes calls the Execute method of the component we are editing, determined using the Component property of the TComponentEditor class. Knowing the type of the component, we can easily access its methods after a dynamic type cast.

The last method refers to the default action of the component and is activated by double-clicking it in the Form Designer:

```
procedure.Edit;
begin
x
  Beep;
  ExecuteVerb (0);
end;
```

Registering the Component Editor

To make this editor available to the Delphi environment, we need to register it. Once more we can add to its unit a `Register` procedure and call a specific registration procedure for component editors:

```
procedure TMdListCompEditor.Edit;
begin
  // produce a beep and show the about box
  Beep;
  ExecuteVerb (0);
end;
```

I've added this unit to the MdDesPk package, which includes all of the design-time extensions of this chapter. After installing and activating this package you can create a new project, place a tabbed list component in it, and experiment with it.

What's Next?

In this chapter we have seen how to define various types of properties, how to add events, and how to define and override component methods. We have seen various examples of components, including simple changes to existing ones, new graphical components, and, in the final section, a dialog box inside a component. While building these components, we have faced some new Windows programming challenges. In general, programmers often need to use the Windows API directly when writing new Delphi components.

Writing components is a very handy technique for reusing software, but to make your components easier to use, you should try to integrate them as much as possible within the Delphi environment, writing property editors and component editors.

There are many more extensions of the Delphi IDE you can write, including custom wizards. I've personally built many Delphi extensions, some of which are available (with source code) on my Web site, `www.marcocantu.com`.

After discussing components and delving a little into the Delphi environment, the next chapter focuses on Delphi DLLs. We have already met DLLs in many previous chapters, and it is time for a detailed discussion of their role and how to build them. In the same chapter, I'll also further discuss the use of Delphi packages, which are a special type of DLL.

Chapter 12

Libraries and Packages

- DLLs in Windows 95, 98, and NT
- Building and using DLLs in Delphi
- Calling DLL functions at run time
- Sharing data in DLLs
- The structure of Delphi packages
- Placing forms in packages and DLLs

Windows executable files come in two flavors: *programs* and *dynamic link libraries* (DLLs). When you write a Delphi application, you typically generate a program file, an EXE. However, Delphi applications often use calls to functions stored in DLLs. Each time you call a Windows API function directly, you actually access a DLL. Delphi also allows programmers to use run-time DLLs for the component library. When you create a package, you basically create a DLL. Delphi can also generate plain dynamic link libraries. The New page of the Object Repository includes a DLL skeleton generator, which generates very few lines of source code.

It is very simple to generate a DLL in the Delphi environment. However, some problems arise from the nature of DLLs. Writing a DLL in Windows is not always as simple as it seems, because the DLL and the calling program need to agree on calling conventions, parameter types, and other details. This chapter covers the basics of DLL programming from the Delphi point of view and provides some simple examples of what you can place in a Delphi DLL. While discussing the examples, I'll also refer to other programming languages and environments, simply because one of the key reasons for writing a procedure in a DLL is to be able to call it from a program written in a different language.

The second part of the chapter will focus on a specific type of dynamic link library, the Delphi *package*. These packages are not as easy to use as they first seem, and it took Delphi programmers some time to figure out how to take advantage of them effectively. Here I'm going to share with you some of these interesting tips and techniques.

The Role of DLLs in Windows

Before delving into the development of DLLs in Delphi and other programming languages, I'll give you a short technical overview of DLLs in Windows, highlighting the key elements. We will start by looking at dynamic linking, then see how Windows uses DLLs, explore the differences between DLLs and executable files, and end with some general rules to follow when writing DLLs.

What Is Dynamic Linking?

First of all, you need to understand the difference between static and dynamic linking of functions or procedures. When a subroutine is not directly available in a source file, the compiler adds the subroutine to an internal table, which includes all external symbols. Of course, the compiler must have seen the declaration of the subroutine and know about its parameters and type, or it will issue an error.

After compilation of a normal—*static*—subroutine, the linker fetches the subroutine's compiled code from a Delphi compiled unit (or static library) and adds it to the executable. The resulting EXE file includes all the code of the program and of the units involved. The Delphi linker is smart enough to include only the minimum amount of code of the units used by the program and to link only the functions and methods that are actually used.

NOTE A notable exception to this rule is the inclusion of virtual methods. The compiler cannot determine in advance which virtual methods the program is going to call, so it has to include them all. For this reason, programs and libraries with too many virtual functions tend to generate larger executable files. While developing the VCL, the Borland developers had to balance the flexibility obtained with virtual functions against the reduced size of the executable files achieved by limiting the virtual functions.

In the case of dynamic linking, which occurs when your code calls a DLL-based function, the linker simply uses the information in the `external` declaration of the subroutine to set up some tables in the executable file. When Windows loads the executable file in memory, first it loads all the required DLLs, and then the program starts. During this loading process, Windows fills the program's internal tables with the addresses of the functions of the DLLs in memory. If for some reason the DLL is not found, the program won't even start, often complaining with nonsense error messages (such as the notorious "a device attached to your system is not functioning").

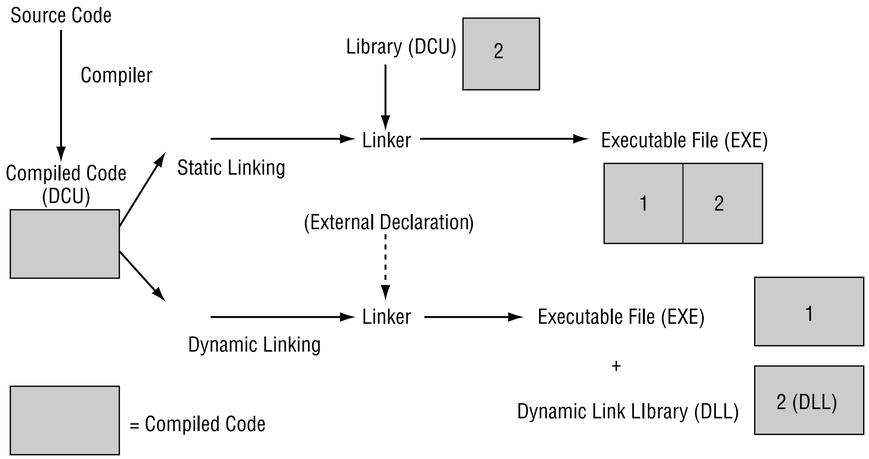
Each time the program calls an external function, it uses this internal table to forward the call to the DLL code (which is now located in the program's address space). Note that this scheme does not involve two different applications. The DLL becomes part of the running program and is loaded in the same address space. All the parameter passing takes place on the application's stack (because the DLL doesn't have a separate stack).

You can see a sketch of how the program calls statically or dynamically linked functions in Figure 12.1. Notice that I haven't yet discussed compilation of the DLL—because I wanted to focus on the two different linking mechanisms first.

NOTE The term *dynamic linking*, when referring to DLLs, has nothing to do with the late-binding feature of object-oriented programming languages. Virtual and dynamic methods in Object Pascal have nothing to do with DLLs. Unfortunately, the same term is used for both kinds of procedures and functions, which causes a lot of confusion. When I speak of dynamic linking in this chapter, I am referring not to polymorphism but to DLL functions.

FIGURE 12.1:

Static and dynamic linking in Windows



There is another approach to using DLLs, which is even more dynamic than the one we have just discussed. In fact, at run time, you can load a DLL in memory, search for a function (provided you know its name), and call the function by name. This approach requires more complex code and takes some extra time to locate the function. The execution of the function, however, has the same speed of the call of an implicitly loaded DLL. On the positive side, you don't need to have the DLL available to start the program. We will use this approach in the DynaCall example later in the chapter.

For the most part, the internal structure of a normal executable file (an EXE file) and a dynamic link library (a DLL, whatever its extension) is the same. They are both executable files. An important difference between programs and DLLs is that a DLL, even when loaded in memory, is not a running program. It is only a collection of procedures and functions that other programs can call. These procedures and functions use the stack of the calling program (the *calling thread*, to be precise). So another difference between a program and a library is that a library doesn't create its own stack—it uses the stack of the program calling it. In Win32, because a DLL is loaded into the application's address space, any memory allocations of the DLL or any global data it creates reside in the address space of the main process.

What Are DLLs For?

Now that you have a general idea of how DLLs work, we can focus on the reasons for using them in Windows:

- If different programs use the same DLL, the DLL is loaded in memory only once, thus saving system memory. DLLs are mapped into the private address space of each process (each running application), but their code is loaded in memory only once.

NOTE The operating system will try to load the DLL at the same address in each application's address space (using the preferred base address specified by the DLL). If that address is not available in a particular application's virtual address space, the DLL code image for that process will have to be relocated, an operation that is expensive in both performance and memory use. The reason is that the relocation happens on a per-process basis, not system-wide.

- You can provide a different version of a DLL, replacing the current one. If the subroutines in the DLL have the same parameters, you can run the program with the new version of the DLL without having to recompile it. If the DLL has new subroutines, it doesn't matter at all. Problems might arise only if a routine in the older version of the DLL is missing in the new one. Problems also arise if the new DLL does not implement the functions in a manner that is compatible with the operation of the old DLL.

These generic advantages apply in several cases. If you have a complex algorithm, or some complex forms required by several applications, you can store them in a DLL. This will let you reduce the executable's size and save some memory when you run several programs using those DLLs at the same time.

The second advantage is particularly applicable to complex applications. If you have a very big program that requires frequent updates and bug fixes, dividing it into several executables and DLLs allows you to distribute only the changed portions instead of one single large executable. This makes sense for Windows system libraries in particular: You generally don't need to recompile your code if Microsoft provides you an updated version of Windows system libraries—for example, in a new version of the operating system.

Another common technique is to use DLLs to store nothing except resources. You can build different versions of a DLL containing strings for different languages and then change the language at run time, or you can prepare a library of icons and bitmaps and then use them in different applications. The development of language-specific versions of a program is particularly important, and Delphi includes support for it through the Integrated Translation Environment (ITE) and the external environment, which are more advanced topics than I have room to go into.

Another key advantage is that DLLs are independent of the programming language. Most Windows programming environments, including most macro languages in end-user applications, allow a programmer to call a subroutine stored in a DLL. This means you can build a DLL in Delphi and call it from Visual Basic, Excel, and many other Windows applications.

Understanding System DLLs

The Windows system DLLs take advantage of all the key benefits of DLLs I've just highlighted. For this reason, it is worth examining them. First of all, Windows has many system DLLs. The three central portions of Windows—Kernel, User, and GDI—are implemented using DLLs (with 32-bit or 16-bit code depending on the OS version). Other system DLLs are operating-system extensions, such as the DLLs for common dialog boxes and controls, OLE, device drivers, fonts, ActiveX controls, and hundreds of others.

Dynamic system libraries are one of the technical foundations of the Windows operating systems. Since each application uses the system DLLs for anything from creating a window to producing output, every program is linked to those DLLs. When you change your printer, you do not need to rebuild your application or get a new version of the Windows GDI library, which manages the printer output. You only need to provide a specific driver, which is a DLL called by GDI to access your printer. Each printer type has its own driver DLL, which makes the system extremely flexible.

From a different point of view, version handling is important for the system itself. If you have an application compiled for Windows 95, you should be able to run it on Windows Me, Windows 2000, and (possibly) future versions of Windows, but the application might behave differently, as each version of Windows has different system code.

The system DLLs are also used as system-information archives. For example, the User DLL maintains a list of all the active windows in the system, and the GDI DLL holds the list of active pens, brushes, icons, bitmaps, and the like. The free memory area of these two system DLLs is usually called "free system resources," and the fact that it is limited plays a very important role in Windows versions still relying on 16-bit code, such as the Windows 9*x* family. On NT platforms, GDI and User resources are limited only by available system memory.

Rules for Delphi DLL Writers

In short, there are some rules for Delphi DLL programmers. A DLL function or procedure to be called by external programs must follow these guidelines:

- It has to be listed in the DLL's `exports` clause. This makes the routine visible to the outside world.
- Exported functions should also be declared as `stdcall`, to use the standard Win32 parameter-passing technique instead of the optimized `register` parameter-passing technique (which is the default in Delphi). The exception to this rule is if you want to use these libraries only from other Delphi applications.

- The types of the parameters of a DLL should be the default Windows types, at least if you want to be able to use the DLL within other development environments. There are further rules for exporting strings, as we'll see in the FirstDll example.
- A DLL can use global data that won't be shared by calling applications. Each time an application loads a DLL, it stores the DLL's global data in its own address space, as we will see in the DllMem example.

Using Existing DLLs

We have already used existing DLLs in examples in this book, when calling Windows API functions. As you might remember, all the API functions are declared in the system Windows unit. Functions are declared in the `interface` portion of the unit, as shown here:

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL; stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints; p4: Integer):
  BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL; stdcall;
```

Then, in the `implementation` portion, instead of providing each function's code, the unit refers to the external definition in a DLL:

```
const
  gdi32 = 'gdi32.dll';

function PlayMetaFile; external gdi32 name 'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name 'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

NOTE In `Windows.PAS` there is a heavy use of the `{$EXTERNALSYM identifier}` directive. This has little to do with Delphi itself; it applies to C++Builder. This symbol prevents the corresponding Pascal symbol from appearing in the C++ translated header file. This helps keep the Delphi and C++ identifiers in synch, so that code can be shared between the two languages.

The external definition of these functions refers to the name of the DLL they use. The name of the DLL must include the .DLL extension, or the program will not work under Windows 2000 (even though it will work under Windows 9*x*). The other element is the name of the DLL function itself. The `name` directive is not necessary if the Pascal function (or procedure) name matches the DLL function name (which is case-sensitive).

To call a function that resides in a DLL, you can provide its declaration and external definition, as shown above, or you can merge the two in a single declaration. Once the function is properly defined, you can call it in the code of your Delphi application just like any other function.

Using a C++ DLL

As an example, I've written a very simple DLL in C++, with some trivial functions, just to show you how to call DLLs from a Delphi application. I won't explain the C++ code in detail (it's basically C code, anyway) but will focus instead on the calls between the Delphi application and the C++ DLL. In Delphi programming it is common to use DLLs written in C or C++.

Suppose you are given a DLL built in C or C++. You'll generally have in your hands a .DLL file (the compiled library itself), an .H file (the declaration of the functions inside the library), and a .LIB file (another version of the list of the exported functions for the C/C++ linker). This LIB file is totally useless in Delphi, while the DLL file is used as-is, and the H file has to be translated into a Pascal unit with the corresponding declarations.

In the following listing, you can see the declaration of the C++ functions I've used to build the CppDll library example. The complete source code and the compiled version of the C++ DLL and of the source code of the Delphi application using it are in the `CppDll` directory on the CD. You should be able to compile this code with any C++ compiler; I've tested it only with recent Borland C++ compilers. Here are the C++ declarations of the functions:

```
extern "C" __declspec(dllexport)
int WINAPI Double (int n);
extern "C" __declspec(dllexport)
int WINAPI Triple (int n);
__declspec(dllexport)
int WINAPI Add (int a, int b);
```

The three functions perform some basic calculations on the parameters and return the result. Notice that all the functions are defined with the `WINAPI` modifier, which sets the proper parameter-calling convention; and they are preceded by the `__declspec(dllexport)` declaration, which makes the functions available to the outside world.

Two of these C++ functions also use the C naming convention (indicated by the `extern "C"` statement), but the third one, `Add`, doesn't. This affects the way we call these functions in Delphi. In fact, the internal names of the three functions correspond to their names in the C++ source code file, except for the `Add` function. Since we didn't use the `extern "C"` clause for this function, the C++ compiler used *name mangling*. This is a technique used to include information about the number and type of parameters in the function name, which the C++ language requires in order to implement function overloading. The result when using the Borland C++ compiler is a funny function name: `@Add$qqsii`. This is actually the name we have to use in our Delphi example to call the `Add` DLL function (which explains why you'll generally avoid C++ name mangling in exported functions, and why you'll generally declare them all as `extern "C"`). The following are the declarations of the three functions in the Delphi CallCpp example:

```
function Add (A, B: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name '@Add$qqsii';
```

```
function Double (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name 'Double';
function Triple (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL';
```

As you can see, you can either provide or omit an alias for an external function. I've provided one for the first function (there was no alternative, because the exported DLL function name `@Add$qqsii` is not a valid Pascal identifier) and for the second, although in the second case it was unnecessary. If the two names match, in fact, you can omit the `name` directive, as I did for the third function above. If you are not sure of the actual names of the functions exported by the DLL, you can use the optional Windows viewer for executable files, with the QuickView command of Windows Explorer or Borland's TDump32 command-line program, available in the Delphi `BIN` folder.

Remember to add the `stdcall` directive to each definition, so that the caller module (the application) and the module being called (the DLL) use the same parameter-passing convention. If you fail to do so, you will get random values passed as parameters, a bug that is very hard to trace.

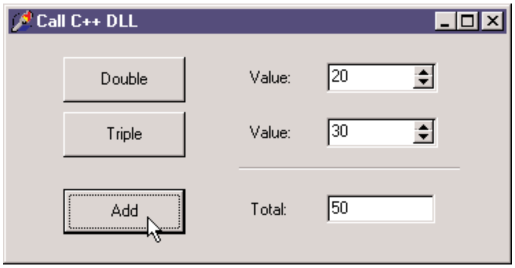
NOTE When you have to convert a large C/C++ header file to the corresponding Pascal declarations, instead of doing a manual conversion you can use a tool to partially automate the process. One of these tools is HeadConv, written by Bob Swart. You'll find a copy on his Web site, `www.drbob42.com`. Notice, though, that automatic header translation from C/C++ to Pascal is not possible, because Pascal is more strongly typed than C/C++, so you have to use types more precisely.

To use this C++ DLL, I've built a Delphi example, named CallCpp. Its simple form has buttons to call the functions of the DLL and some visual components for input and output parameters (see Figure 12.2). The code of the button event handlers looks like:

```
procedure TForm1.BtnDoubleClick(Sender: TObject);
begin
  SpinEdit1.Value := Double (SpinEdit1.Value);
end;
```

FIGURE 12.2:

The output of the CallCpp example when you have pressed each of the buttons



Value: 20

Value: 30

Total: 50

Notice that to run this application, you should have the DLL in the same directory as the project, in one of the directories on the path, or in the Windows or System directories. If you move the executable file to a new directory and try to run it, you'll get a run-time error indicating that the DLL is missing.

Creating a DLL in Delphi

Besides using DLLs written in other environments, you can use Delphi to build DLLs that can be used by Delphi programs or with any other development tool that supports DLLs. Building DLLs in Delphi is so easy that you might overuse this feature. In general, I suggest you try to build components and packages instead of plain DLLs. As I'll discuss later in this chapter, packages often contain components, but they can also include plain noncomponent classes, allowing you to write object-oriented code and to reuse it effectively.

Placing a collection of functions in a DLL is a more traditional approach to programming: DLLs cannot export classes and objects, at least unless you use Microsoft's COM technology or some other advanced techniques.

As I've already mentioned, building a DLL is useful when a portion of the code of a program is subject to frequent changes. In this case you can frequently replace the DLL, keeping the rest of the program unchanged. Similarly, when you need to write a program that provides different features to different groups of users, you can distribute different versions of a DLL to those users.

A Simple Delphi DLL

As a starting point in exploring the development of DLLs in Delphi, I'll show you a very simple library built in Delphi. The primary focus of this example will be to show the syntax you use to define a DLL in Delphi, but it will also illustrate a few considerations involved in passing string parameters. To start, select the File > New > Other command and choose the DLL option in the New page of the Object Repository. This creates a very simple source file that starts with the following definition:

```
library Project1;
```

The library statement indicates that we want to build a DLL instead of an executable file. Now we can add routines to the library and list them in an exports statement:

```
function Triple (N: Integer): Integer; stdcall;
begin
  Result := N * 3;
end;
```

```
function Double (N: Integer): Integer; stdcall;
begin
  Result := N * 2;
end;

exports
  Triple, Double;
```

In this basic version of the DLL, we don't need a `uses` statement; but in general, the main project file includes only the `exports` statement, while the function declarations are placed in a separate unit. In the final source code of the FirstDll example on the CD, I've actually changed the code slightly from the version listed above, to show a message each time a function is called. There are two ways to accomplish this. The simplest is to use the Dialogs unit and call the ShowMessage function.

The code requires Delphi to link a lot of VCL code into the application. If you statically link the VCL into this DLL, the resulting size will be about 375 KB. The reason is that the `ShowMessage` function displays a VCL form that contains VCL controls and uses VCL graphics classes; and those indirectly refer to things like the VCL streaming system and the VCL application and screen objects. For this simple case, a better alternative is to show the messages using direct API calls, using the Windows unit and calling the MessageBox function, so that the VCL code is not required. This change in code brings the size of the application down to only about 40 KB.

NOTE This huge difference in size underlines the fact that you should not overuse DLLs in Delphi, to avoid compiling the code of the VCL in multiple executable files. Of course, you can reduce the size of a Delphi DLL by using run-time packages, as detailed later in this chapter.

If you run a test program like the CallFrst example (described later) using the API-based version of the DLL, its behavior won't be correct. In fact, you can click the buttons that call the DLL functions several times without first closing the message boxes displayed by the DLL. This happens because the first parameter of the `MessageBox` API call above is zero. Its value should instead be the handle of the program's main form or the application form, information I don't have at hand in the DLL itself.

Overloaded Functions in Delphi DLLs

When you create a DLL in C++, overloaded functions use name mangling to generate a different name for each function, including the type of the parameters right in the name, as we've seen in the CppDll example.

When you create a DLL in Delphi and use overloaded functions (that is, multiple functions using the same name and marked with the overload directive), Delphi allows you to export only one of the overloaded functions with the original name, indicating its parameters list in the exports clause. If you want to export multiple overloaded functions, you should specify different names in the exports clause to distinguish the overloads. This is demonstrated by this portion of the FirstDLL code:

```
function Triple (C: Char): Integer; stdcall; overload;
function Triple (N: Integer): Integer; stdcall; overload;

exports
  Triple (N: Integer),
  Triple (C: Char) name 'TripleChar';
```

NOTE The reverse is possible as well: You can import a series of similar functions from a DLL and define them all as overloaded functions in the Pascal declaration. Delphi's OpenGL.PAS unit contains a series of examples of this technique.

Exporting Strings from a DLL

In general, functions in a DLL can use any type of parameter and return any type of value. There are two exceptions to this rule:

- If you plan to call the DLL from other programming languages, you should probably try using Windows native data types instead of Delphi-specific types. For example, to express color values, you should use integers or the Windows ColorRef type instead of the Delphi native TColor type, doing the appropriate conversions (as in the FormDLL example, described in the next section). Other Delphi types that, for compatibility, you should avoid using include objects, which cannot be used by other languages at all, and Pascal strings, which can be replaced by PChar strings. In other words, every Windows development environment must support the basic types of the API, and if you stick to them, your DLL will be usable with other development environments. Also, Pascal file variables (text files and binary file of record) should not be passed out of DLLs, but you can use Win32 file handles.
- Even if you plan to use the DLL only from a Delphi application, you cannot pass Delphi strings (and dynamic arrays) across the DLL boundary without taking some precautions. This is because of the way Delphi manages strings in memory—allocating, reallocating, and freeing them automatically. The solution to the problem is to include the ShareMem system unit both in the DLL and in the program using it. This unit must be included as the first unit of each of the projects.

NOTE Objects actually can be passed out of a DLL, if the objects are designed to be used like interfaces or pure abstract classes. All methods of these objects must be virtual, and objects must be created by the DLL. This is more or less what happens with COM objects, the approach you should use for multilanguage applications. For Delphi-only projects with libraries exporting objects, you should rather use packages, as we'll see later in this chapter.

In the FirstDLL example, I've actually included both approaches: One function receives and returns a Pascal string, and another one receives as parameter a PChar pointer, which is then filled by the function itself. The first function is very simple:

```
function DoubleString (S: string; Separator: Char): string; stdcall;
begin
  Result := S + Separator + S;
end;
```

The second one is quite complex because PChar strings don't have a simple + operator, and they are not directly compatible with characters; the separator must be turned into a string before adding it. Here is the complete code; it uses input and output PChar buffers, which are compatible with any Windows development environment:

```
function DoublePChar (BufferIn, BufferOut: PChar;
  BufferOutLen: Cardinal; Separator: Char): LongBool; stdcall;
var
  SepStr: array [0..1] of Char;
begin
  // if the buffer is large enough
  if BufferOutLen > StrLen (BufferIn) * 2 + 2 then
  begin
    // copy the input buffer in the output buffer
    StrCopy (BufferOut, BufferIn);
    // build the separator string (value plus null terminator)
    SepStr [0] := Separator;
    SepStr [1] := #0;
    // append the separator
    StrCat (BufferOut, SepStr);
    // append the input buffer once more
    StrCat (BufferOut, BufferIn);
    Result := True;
  end
  else
    // not enough space
    Result := False;
end;
```

This second version of the code is certainly more complex, but the first can be used only from Delphi. Moreover, the first version requires us to include the ShareMem unit in the DLL (and in the programs using it) and to deploy the file BorlndMM.DLL (the name stands for Borland Memory Manager) along with the program and the specific library.

Calling the Delphi DLL

How can we use the library we've just built? We can call it from within another Delphi project or from other environments. As an example, I've built the CallFrst project (stored in the `FirstDLL` directory).

To access the DLL functions, we must declare them as `external`, as we've done with the C++ DLL. This time, however, we can simply copy and paste the definition of the functions from the source code of the Delphi DLL, adding the `external` clause, as in:

```
function Double (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';
```

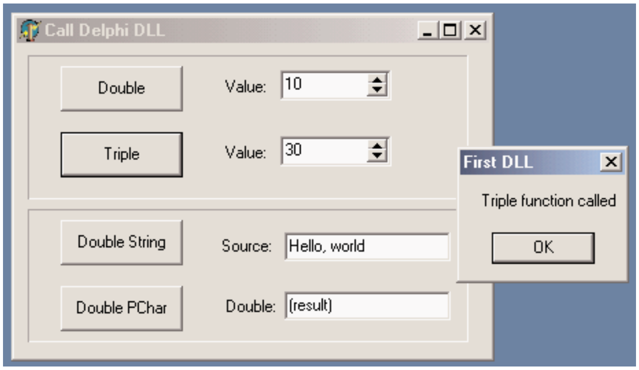
This declaration is similar to those used to call the C++ DLL. This time, however, we have no problems with function names. The source code of the example is actually quite simple. Once they are redeclared as `external`, the functions of the DLL can simply be used as if they were local functions. Here are two examples, with calls to the string-related functions:

```
procedure TForm1.BtnDoubleStringClick(Sender: TObject);
begin
  // call the DLL function directly
  EditDouble.Text := DoubleString (EditSource.Text, ';');
end;

procedure TForm1.BtnDoublePCharClick(Sender: TObject);
var
  Buffer: string;
begin
  // make the buffer large enough
  SetLength (Buffer, 1000);
  // call the DLL function
  if DoublePChar (PChar (EditSource.Text), PChar (Buffer), 1000, '/') then
    EditDouble.Text := Buffer;
end;
```

Figure 12.3 shows the effect of this program's calls to the DLL.

FIGURE 12.3:

The output of the CallFrst example, which calls the DLL we've built in Delphi



10

30

Hello, world

(result)

Triple function called

Project and Library Names in Delphi 6

For a library, as for a standard application, you end up with a library name matching a Delphi project filename. Following a similar technique introduced in Kylix for compatibility with standard Linux naming conventions for shared object libraries (the Linux equivalent of Windows DLLs), Delphi 6 introduced special compiler directives you can use in libraries to determine their executable filename. Some of these directives make more sense in the Linux world than on Windows, but they've all been added anyway.

- `$LIBPREFIX` is used to add something in front of the library name. Paralleling the Linux technique of adding *lib* in front of library names, this directive is used by Kylix to add *bpl* at the beginning of package names. This is due to the fact that Linux uses a single extension (`.so`) for libraries, while in Windows you can have different library extensions, something Borland uses for packages (`.bpl`).
- `$LIBSUFFIX` is used to add text after the library name and before the extension. This can be used to specify versioning information or other variations on the library name and can be quite useful also on Windows.
- `$LIBVERSION` is used to add a version number after the extension—something very common in Linux, but you should generally avoid this on Windows.

As an example, consider the following directives, which generate a library called `MarcoNameTest60.dll`:

```
library NameTest;
{$LIBPREFIX 'Marco'}
{$LIBSUFFIX '60'}
```

NOTE Unlike past versions, Delphi 6 packages use the `$LIBSUFFIX` directive extensively. For this reason, the VCL package generates the `VCL.DCP` file and the `VCL60.BPL` file. The advantage of this approach is that you won't need to change the `requires` portions of your packages for every new version of Delphi. Of course, this will become handy to maintain Delphi 7– and Delphi 6–compatible packages, but isn't helpful now for Delphi 5 compatibility.

A Delphi Form in a DLL

Besides writing simple DLLs with functions and procedures, you can place a complete form built with Delphi into a DLL. This can be a dialog box or any other kind of form, and it can be used not only by other Delphi programs, but also by other development environments or macro languages.

To build the FormDLL example, I've built a simple form with three scroll bars you can use to select a color and two preview areas for the resulting pen and brush colors. The form also contains two bitmap buttons and has its `BorderStyle` property set to bsDialog. Aside from developing a form as usual, I've only added two new subroutines to the unit that defines the form. In the `interface` portion of the unit, I've added the following declarations:

```
function GetColor (Col: LongInt): LongInt; stdcall;
procedure ShowColor (Col: LongInt;
  FormHandle: THandle; MsgBack: Integer); stdcall;
```

In both subroutines the `Col` parameter is the initial color. Notice that I've passed it as a long integer, which corresponds to the Windows ColorRef data type. As mentioned before, using the `TColor` Delphi type might have caused problems with non-Delphi applications: Even though a `TColor` is very similar to a ColorRef, these types don't always correspond. When you write a DLL, I suggest you use only the Windows native data types (unless you are sure only Delphi programs will use the DLL).

The `GetColor` function returns the final color (which is the same as the initial color if the user clicks the Cancel button). The value is returned immediately because the function shows the form as a modal form. The `ShowColor` procedure, instead, simply displays the form (as a modeless form) and returns immediately. For this reason the form needs a way to communicate back to the calling form. In this case I've decided to pass as parameters the handle for the window of the calling form and the ID of the message to use to communicate back with it.

In the next sections, you'll see how to write the code of the two subroutines; and you'll also see what problems arise, particularly when you place a modeless form in a DLL. Of course, I'll also provide a few alternative fixes.

Using the DLL Form as Modal

When you want to place a Delphi component (such as a form) in a DLL, you can only provide functions that create, initialize, or run the component or access its properties and data. The simplest approach is to have a single function that sets the data, runs the component, and returns the result, as in the modal version. Here is the code of the function, added to the `implementation` portion of the unit that defines the form:

```
function GetColor (Col: LongInt): LongInt; stdcall;
var
  FormScroll: TFormScroll;
begin
  // default value
  Result := Col;
  try
    FormScroll := TFormScroll.Create (Application);
    try
```

```
        // initialize the data
        FormScroll.SelectedColor := Col;
        // show the form
        if FormScroll.ShowModal = mrOK then
          Result := FormScroll.SelectedColor;
      finally
        FormScroll.Free;
      end;
    except
      on E: Exception do
        MessageDlg ('Error in FormDLL: ' + E.Message, mtError, [mbOK], 0);
    end;
  end;
```

An important element is the structure of the GetColor function. The code creates the form at the beginning, sets some initial values, and then runs the form, eventually extracting the final data. What makes this different from the code we generally write in a program is the use of exception handling:

- A try/except block protects the whole function, so that any exception generated by the function will be trapped, displaying a proper message. The reason for handling every possible exception is that the calling application might be written in any language, in particular one that doesn't know how to handle exceptions. Even when the caller is a Delphi program, it is sometimes useful to use the same protective approach.
- A try/finally block protects the operations on the form, ensuring that the form object will be properly destroyed, even when an exception is raised.

By checking the return value of the ShowModal method, the program determines the result of the function. I've set the default value before entering the try block to ensure that it will always be executed (and also to avoid the compiler warning indicating that the result of the function might be undefined).

Now that we have updated the form and written the code of the unit, we can move to the project source code, which (temporarily) becomes the following:

```
library FormDLL;

uses
  ScrollF in 'SCROLLF.PAS' {FormScroll};

exports
  GetColor;
end.
```

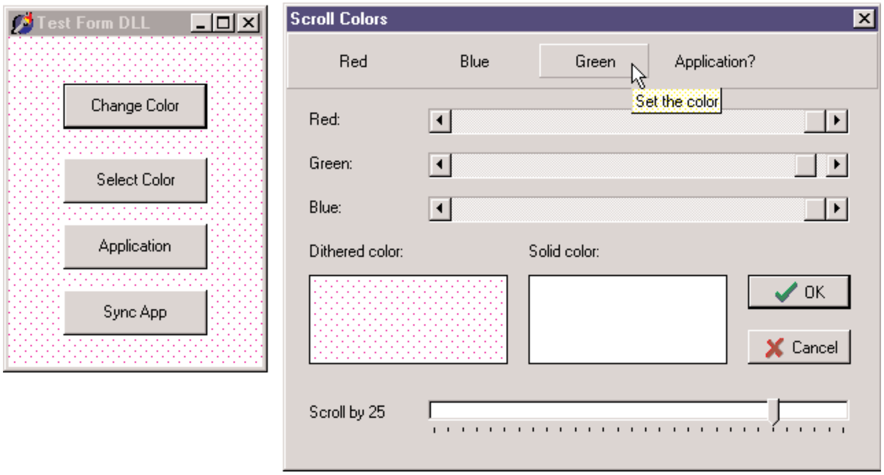
We can now use a Delphi program to test the form we have placed in the DLL. The UseCol example is in the same directory as the previous DLL, FormDLL (and both projects are part of the FormDLL project group, the file `FormDll.BPG`). The form of the UseCol example contains a button to call the `GetColor` function of the DLL. Here is the definition of this function and the code of the `Button1Click` method:

```
function GetColor (Col: LongInt): LongInt; stdcall; external 'FormDLL.DLL';

procedure TForm1.Button1Click(Sender: TObject);
var
  Col: LongInt;
begin
  Col := ColorToRGB (Color);
  Color := GetColor (Col)
end;
```

Running this program (see Figure 12.4) displays the dialog box, using the current background color of the main form. If you change the color and click OK, the program uses the new color as the background color for the main form.

FIGURE 12.4:

The execution of the UseCol test program when it calls the dialog box we have placed in the FormDLL



If you execute this as a modal dialog box, almost all the features of the form work fine. You can see the tooltips, the flat speed buttons in the toolbar behave properly, and you get no extra entry in the task bar. This might be obvious, but is not what will happen when we use the form inside the DLL as a modeless form. Even with modal forms, however, I recommend synchronizing the application objects of the DLL and the executable file, as described in the next section.

A Modeless Form in a DLL

The second subroutine of the FormDLL example uses a different approach. As mentioned, it receives three parameters: the color, the handle of the main form, and the message number for notification when the color changes. These values are stored in the private data of the form:

```
procedure ShowColor (Col: LongInt;
  FormHandle: THandle; MsgBack: Integer); stdcall;
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  try
    // initialize the data
    FormScroll.FormHandle := FormHandle;
    FormScroll.MsgBack := MsgBack;
    FormScroll.SelectedColor := Col;
    // show the form
    FormScroll.Show;
  except
    on E: Exception do
    begin
      MessageDlg ('Error in FormDLL: ' + E.Message, mtError, [mbOK], 0);
      FormScroll.Free;
    end;
  end;
end;
```

When the form is activated, it checks to see if it was created as a modal form (simply testing the FormHandle field). In this case, the form changes the caption and the behavior of the OK button, as well as the overall style of the Cancel button (you can see the modified buttons in Figure 12.5):

```
procedure TFormScroll.FormActivate(Sender: TObject);
begin
  // change buttons for modeless form
  if FormHandle <> 0 then
  begin
    BitBtn1.Caption := 'Apply';
    BitBtn1.OnClick := ApplyClick;
    BitBtn2.Kind := bkClose;
  end;
end;
```

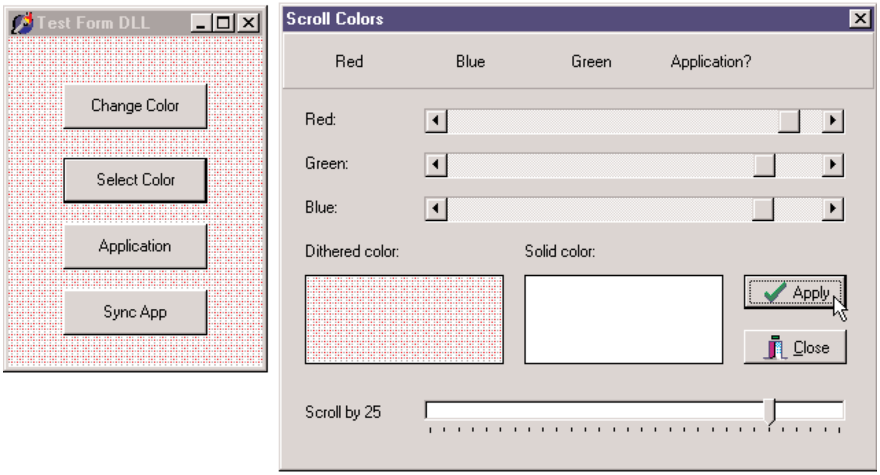


FIGURE 12.5: When the DLL-based form is used as a modeless form, its buttons are slightly modified (as you can see comparing this image with that of Figure 12.4).

The `ApplyClick` method I've manually added to the form simply sends the notification message to the main form, using one of the parameters to send back the selected color:

```
SendMessage (FormHandle, MsgBack, SelectedColor, 0);
```

Finally, the form's `OnClose` event destroys the form object, by setting the `Action` parameter to caFree. Now let's move back to the demo program. The second button of the UseForm example's form has the following code:

```
procedure TForm1.BtnSelectClick(Sender: TObject);
var
  Col: LongInt;
begin
  Col := ColorToRGB (Color);
  ShowColor (Col, Handle, wm_user);
end;
```

The form also has a message-handling method, connected with the `wm_user` message. This method reads the value of the parameter corresponding to the color and sets it:

```
procedure TForm1.UserMessage(var Msg: TMessage);
begin
  Color := Msg.WParam;
end;
```

Running this program produces some strange effects. Basically, the modeless form and the main form are not synchronized, so they both show up in the Windows Taskbar; and when you minimize the main form, the other one remains on the screen. The two forms behave as if they were part of separate applications, because two Delphi programs (the DLL and the EXE) have two separate global `Application` objects, and only the `Application` object of the executable file has an associated window.

To test this situation, I've added a button to both the main form and the DLL form, showing the numeric value of the Application object's handle. Here is the code for one of them:

```
ShowMessage ('Application Handle: ' + IntToStr (Application.Handle));
```

For the form in the DLL, you'll invariably get the value 0, while for the form in the executable you get a numeric value determined each time by Windows.

To fix the problem we can add to the DLL an initialization function that passes the handle of the application window to the library. In practice, we copy the Handle of the executable's Application object to the same property of the DLL's Application object. This is enough to synchronize the two Application objects and make the two forms behave as in a simple Delphi program. Here is the code of the function in the DLL:

```
procedure SyncApp (AppHandle: THandle); stdcall;
begin
  Application.Handle := AppHandle;
end;
```

And here is the call to it in the executable file:

```
procedure TForm1.BtnSyncClick(Sender: TObject);
begin
  SyncApp (Application.Handle);
  BtnSync.Enabled := False;
end;
```

NOTE Assigning the handle of the application object of the DLL is not a work-around for a bug but a documented operation required by the VCL. The VCL Application object supports assignment to its Handle property (unlike most other Handle properties of the VCL) specifically to allow programmers to tie DLL-based forms into the environment of a host application.

I've connected this code to a button, instead of executing it automatically at startup, to let you test the behavior in the two different cases. Before you click the Sync App button, the secondary modeless form behaves oddly. If you close it, synchronize the applications, and then create another instance of the modeless form, it will behave almost correctly. The only visible problem is that the flat speed buttons of the modeless form won't be highlighted when the mouse moves over them. We'll see how to fix this problem using run-time packages at the end of the chapter.

NOTE Technically this behavior of the speed buttons depends on the fact that the controls in the DLL form don't receive the cm_MouseEnter and cm_MouseLeave messages, because the DLL's Application.Idle method is never called. The DLL's Application object, in fact, is not running the application's message loop. You can activate it by exporting from the DLL a function that calls the internal Application.Idle routine, and call that function from the host application when its message loop goes idle. As I've mentioned, however, all these problems (and a few others) can be better solved by using run-time packages.

Calling a Delphi DLL from Visual Basic for Applications

We can also display this color dialog box from other programming languages. Calling this DLL from C or C++ is easy. To link the application, you need to generate an import library (using the `IMPLIB` command-line utility) and add the resulting LIB file to the project. Since I've already used a C++ compiler in this chapter, this time I will write a similar example using Microsoft Word for Windows and Visual Basic for Applications instead.

To start, open Microsoft Word. Then open its Macro dialog box (with the Tools ➢ Macro menu item or a similar command, depending on your version of Word), type a new macro name, such as **DelphiColor**, and click the Create button. You can now write the BASIC code, which declares the function of our DLL and calls it. The BASIC macro uses the result of the DLL function in two ways. By calling `Insert`, it adds to the current document a description of the color with the amount of red, green, and blue; and by calling `Print` it displays the numeric value in the status bar:

```
Declare Function GetColor Lib "FormDLL"(Col As Long) As Long
Sub MAIN
  NewColor = GetColor(0)
  Print "The code of the color is " + Str$(NewColor)
  Insert "Red:" + Str$(NewColor Mod 256) + Chr$(13)
  Insert "Green:" + Str$(Int(NewColor / 256) Mod 256) + Chr$(13)
  Insert "Blue:" + Str$(Int(NewColor / (256 * 256))) + Chr$(13)
End Sub
```

Unfortunately, there is no easy way to use RGB colors in Word, since Word's color schemes are based on fixed color codes. Here is an example of the output of this macro:

```
Red: 141
Green: 109
Blue: 179
```

You can find the text of this macro in the file `WORDCALL.TXT`, in the directory containing this DLL. If you want to test it, remember to first copy the DLL file into one of the directories of the path or into the Windows system directory.

NOTE A better way to integrate Delphi code with Office applications is to use OLE Automation, instead of writing custom DLLs and calling them from the macro language. We'll see examples of OLE Automation in Chapter 20.

Calling a DLL Function at Run Time

Up to now, we've always referenced in our code the functions exported by the libraries, so that the DLLs will be loaded along with the program. I mentioned earlier that we can also delay the loading of a DLL until the moment it is actually needed, so we'd be able to use the rest of the program in case the DLL is not available.

Dynamic loading of a DLL in Windows is accomplished by calling the LoadLibrary API function, which searches the DLL in the program folder, in the folders on the path, and in some system folders. If the DLL is not found, Windows will show an error message, something you can skip by calling Delphi's SafeLoadLibrary function. This function has the same effect as the API it encapsulates, but it suppresses the standard Windows error message and should be the preferred way to load libraries dynamically in Delphi.

If the library is found and loaded (something you know by checking the return value of LoadLibrary or SafeLoadLibrary), a program can call the GetProcAddress API function, which searches the DLL's exports table, looking for the name of the function passed as a parameter. If GetProcAddress finds a match, it returns a pointer to the requested procedure. Now we can simply cast this function pointer to the proper data type and call it.

Whichever loading functions you've used, don't forget to call FreeLibrary at the end, so that the DLL can be properly released from memory. In fact, the system uses a reference-counting technique for libraries, releasing them when each loading request has been followed by a freeing request.

The example I've built to show dynamic DLL loading is named DynaCall and uses the FirstDLL library we built earlier in this chapter (to make the program work, I've copied the DLL into the same folder as the DynaCall example). Instead of declaring the Double and Triple functions and using them directly, this example obtains the same effect with somewhat more complex code. The advantage, however, is that the program will run even without the DLL. Also, if new *compatible* functions are added to the DLL, we won't have to revise the program's source code and recompile it to access those new functions. Here is the core code of the program:

```
type
  TIntFunction = function (I: Integer): Integer; stdcall;

const
  DllName = 'Firstdll.dll';

procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFunct: TIntFunction;
begin
  HInst := SafeLoadLibrary (DllName);
  if HInst > 0 then
  try
    FPointer := GetProcAddress (HInst,
      PChar (Edit1.Text));
      if FPointer <> nil then
      begin
        MyFunct := TIntFunction (FPointer);
        SpinEdit1.Value := MyFunct (SpinEdit1.Value);
      end
      else
        ShowMessage (Edit1.Text + ' DLL function not found');
    finally
      FreeLibrary (HInst);
    end
    else
      ShowMessage (DllName + ' library not found');
  end;
```

How do you call a procedure in Delphi, once you have a pointer to it? One solution is to convert the pointer to a procedural type and then call the procedure using the procedural-type variable, as in the listing above. Notice that the procedural type you define must be compatible with the definition of the procedure in the DLL. This is the Achilles' heel of this method—there is no check of the parameter types.

What is the advantage of this approach? In theory, you can use it to access any function of any DLL at any time. In practice, it is useful when you have different DLLs with compatible functions or a single DLL with several compatible functions, as in our case. What we can do is to call the `Double` and `Triple` methods simply by entering their names in the edit box. Now, if someone gives us a DLL with a new function receiving an integer as a parameter and returning an integer, we can call it simply by entering its name in the edit box. We don't even need to recompile the application.

With this code, the compiler and the linker ignore the existence of the DLL. When the program is loaded, the DLL is not loaded immediately. We might make the program even more flexible and let the user enter the name of the DLL to use. In some cases, this is a great advantage. A program may switch DLLs at run time, something the direct approach does not allow. Note that this approach to loading DLL functions is common in macro languages and is used by many visual programming environments. Also, the code of the Word macro we saw earlier in this chapter uses this approach to load the DLL and to call the external function. Well, you don't want to recompile Word, do you?

Only a system based on a compiler and a linker, such as Delphi, can use the direct approach, which is generally more reliable and also a little bit faster. I think the indirect loading approach of the DynaCall example is useful only in special cases, but it can be extremely powerful.

A DLL in Memory: Code and Data

We can use this technique, based on the GetProcAddress API function, to test which memory address of the current process a function has been mapped to, with the following code:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  HDLLInst: THandle;
begin
  HDLLInst := SafeLoadLibrary ('dllmem');
  Label1.Caption := Format ('Address: %p', [
    GetProcAddress (HDLLInst, 'SetData')]);
  FreeLibrary (HDLLInst);
end;
```

This code displays, in a label, the memory address of the function, within the address space of the calling application. If you run two programs using this code, they'll generally both show the same address. This demonstrates that the code is loaded only once at a common memory address.

NOTE The memory address will be different if the DLL had to be relocated in one of the processes, or if each process has relocated the DLL to a different base address. In these cases, the code is generally not shared in memory but actually loaded multiple times, because absolute address references in the code must be rewritten to refer to the proper new addresses. As the code is modified by the loader, it cannot be shared.

If the code of the DLL is loaded only once, what about the global data? Basically, each copy of the DLL has its own copy of the data, in the address space of the calling application. However, it is indeed possible to share global data between applications using a DLL. The most common technique for sharing data is to use memory-mapped files. I'll use this technique for a DLL, but it can also be used to share data directly among applications.

This example is called DllMem and uses a project group with the same name, as in the previous examples of this chapter. The DllMem project group includes the DllMem project (the DLL itself) and the UseMem project (the demo application). The DLL code has a simple project file, which exports four subroutines:

```
library dllmem;

uses
  SysUtils,
  DllMemU in 'DllMemU.pas';

exports
  SetData, GetData,
  GetShareData, SetShareData;
end.
```

The actual code is in the secondary unit (DllMemU.PAS), which has the code of the four routines that read or write two global memory locations. These hold an integer and a pointer to an integer. Here are the variable declarations and the two Set routines:

```
var
  PlainData: Integer = 0; // not shared
  ShareData: ^Integer; // shared

procedure SetData (I: Integer); stdcall;
begin
  PlainData := I;
end;

procedure SetShareData (I: Integer); stdcall;
begin
  ShareData^ := I;
end;
```

Sharing Data with Memory-Mapped Files

For the data that isn't shared, there isn't anything else to do. To access the shared data, however, the DLL has to create a memory-mapped file and then get a pointer to this memory area. These two operations require two Windows API calls:

- CreateFileMapping requires as parameters the filename (or $FFFFFFFF to use a virtual file in memory), some security and protection attributes, the size of the data, and an internal name (which must be the same to share the mapped file from multiple calling applications).
- MapViewOfFile requires as parameters the handle of the memory mapped file, some attributes and offsets, and the size of the data (again).

Here is the source code of the initialization section, executed every time the DLL is loaded into a new process space (that is, once for each application that uses the DLL):

```
var
  hMapFile: THandle;

const
  VirtualFileName = 'ShareDllData';
  DataSize = sizeof (Integer);

initialization
  // create memory mapped file
  hMapFile := CreateFileMapping ($FFFFFFFF, nil,
    Page_ReadWrite, 0, DataSize, VirtualFileName);
  if hMapFile = 0 then
    raise Exception.Create ('Error creating memory-mapped file');
```

```
// get the pointer to the actual data
ShareData := MapViewOfFile (
  hMapFile, File_Map_Write, 0, 0, DataSize);
```

When the application terminates and the DLL is released, it has to free the pointer to the mapped file and the file mapping itself:

```
finalization
  UnmapViewOfFile (ShareData);
  CloseHandle (hMapFile);
```

The code of the program using this DLL, UseMem, is very simple. The form of this application has four edit boxes (two with an UpDown control connected), five buttons, and a label. The first button saves the value of the first edit box in the DLL data, getting the value from the connected UpDown control:

```
SetData (UpDown1.Position);
```

If you click the second button, the program copies the DLL data to the second edit box:

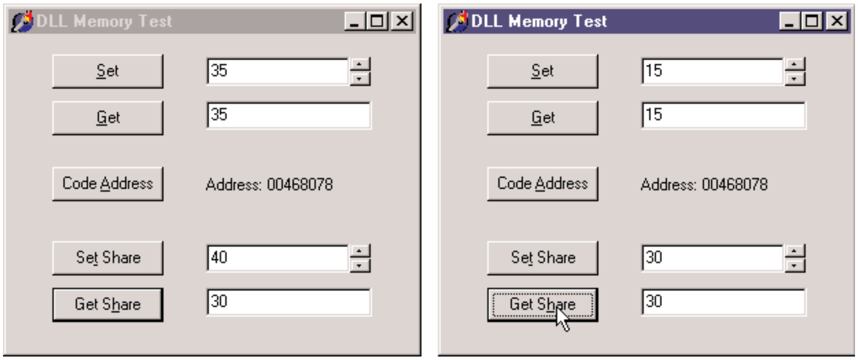
```
Edit2.Text := IntToStr(GetData);
```

The third button is used to display the memory address of a function, with the source code shown at the beginning of this section, and the last two buttons have basically the same code as the first two, but they call the `SetShareData` procedure and `GetShareData` function.

If you run two copies of this program, you can see that each copy has its own value for the plain global data of the DLL, while the value of the shared data is common. Set different values in the two programs and then get them in both, and you'll see what I mean. This situation is illustrated in Figure 12.6.

FIGURE 12.6:

If you run two copies of the UseMem program, you'll see that the global data in its DLL is not shared.



35
35
Address: 00468078
40
30

15
15
Address: 00468078
30
30

WARNING Memory-mapped files reserve a minimum of a 64 KB range of virtual addresses and consume physical memory in 4 KB pages. The example's use of 4-byte Integer data in shared memory is rather expensive, especially if you use the same approach for sharing multiple values. If you need to share several variables, you should place them all in a single shared memory area (accessing the different variables using pointers or building a record structure for all of them).

Using Delphi Packages

In Delphi, component packages are an important type of DLL. Packages allow you to bundle a group of components and then link the components either statically (adding their compiled code to the executable file of your application) or dynamically (keeping the component code in a DLL, the run-time package that you'll distribute along with your program). In the last chapter, you saw how to build a package. Now I want to underline some advantages and disadvantages of the two forms of linking for a package. There are many elements to keep in mind:

- Using a package as a DLL makes the executable files much smaller.
- Linking the package units into the program allows you to distribute only part of the package code. Generally, the size of the executable file of an application plus the size of the required package DLLs that it requires is much bigger than the size of the statically linked program. The linker includes only the code actually used by the program, whereas a package must link in all the functions and classes declared in the interface sections of all the units contained in the package.
- If you distribute several Delphi applications based on the same packages, you might end up distributing less code, because the run-time packages are shared. In other words, once the users of your application have the standard Delphi run-time packages, you can ship them very small programs.
- If you run several Delphi applications based on the same packages, you can save some memory space at run time; the code of the run-time packages is loaded in memory only once between the multiple Delphi applications.
- Don't worry too much about distributing a large executable file. Keep in mind that when you make minor changes to a program, you can use any of various tools to create a *patch file*, so that you distribute only a file containing the differences, not a complete copy of the files.
- If you place a few of your program's forms in a run-time package, you can share them among programs. When you modify these forms, however, you'll generally need to recompile the main program as well, and distribute both of them again to your users. The next section discusses this complex topic in detail.

Package Versioning

A very important and often misunderstood element is the distribution of updated packages. When you update a DLL, you can ship the new version, and the executable programs requiring this DLL will generally still work (unless you've removed existing exported functions or changed some of their parameters).

When you distribute a Delphi package, however, if you update the package and modify the interface portion of any unit of the package, you might need to recompile all the applications that use the package. This is required if you add methods or properties to a class, but not if you add new global symbols (or modify anything not used by client applications). There is no problem at all for changes affecting only the implementation section of the package's units.

A DCU file in Delphi has a version tag based on its timestamp and a checksum computed from the interface portion of the unit. When you change the interface portion of a unit, every other unit based on it should be recompiled. The compiler compares the timestamp and checksum of the unit of previous compilations with the new timestamp and checksum, and decides whether the dependent unit must be recompiled. This is why you have to recompile each unit when you get a new version of Delphi, which has modified system units.

A package is a collection of units. In Delphi 3, a checksum of the package, obtained from the checksum of the units it contains and the checksum of the packages it requires, was added as an extra entry function to the package library, so that any executable based on an older version of the package would fail at startup.

Delphi 4 and following versions have relaxed the run-time constraints of the package. The design-time constraints on DCU files remain identical, though. The checksum of the packages is not checked anymore, so you can directly modify the units that are part of a package and deploy a new version of the package to be used with the existing executable file. Since methods are referenced by name, you cannot remove any existing method. You cannot even change its parameters, because of name-mangling techniques specifically added to the packages to protect against changes in parameters.

Removing a method referenced from the calling program will stop the program during the loading process. If you make other changes, however, the program might fail unexpectedly during its execution. For example, if you replace a component placed on a form compiled in a package with a similar component, the calling program might still able to access the one in that memory location, although it is now a different component!

If you decide to follow this treacherous road of changing the interface of units in a package without recompiling all the programs that use it, you should at least limit your changes. When you add new properties or nonvirtual methods to the form, you should be able to maintain full compatibility with existing programs already using the package. Also, adding fields and virtual methods might affect the internal structure of the class, leading to problems with existing programs that expect a different class data and virtual method table (VMT) layout. Of course, this applies to the binary compatibility between the EXE and the BPL (Borland Package Library).

WARNING Here I'm referring to the distribution on compiled programs divided between EXE and packages, not to the distribution of components to other Delphi developers. In this latter case the versioning rules are more stringent, and you must take extra care in package versioning.

Having said this, I recommend never changing the interface of any unit exported by your packages. To accomplish this, you can add to your package a unit with form-creation functions (as in the DLL with forms presented earlier) and use it to access another unit, which defines the form. Although there is no way to *hide* a unit that is linked into a package, if you never directly use the class defined in a unit, but use it only through other routines, you'll have more flexibility in modifying it. You can also use form inheritance to modify a form within a package without really affecting the original version.

The most stringent rule for packages is the following one used by component writers: For long-term deployment and maintenance of code in packages, plan on having a major release with minor maintenance releases. A major release of your package will require all client programs to be recompiled from source; the package file itself should be renamed with a new version number, and the interface sections of units can be modified. Maintenance releases of that package should be restricted to implementation changes to preserve full compatibility with existing executables and units.

Forms Inside Packages

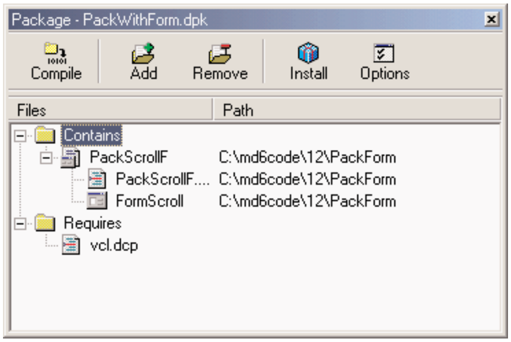
We've already discussed (in Chapter 11, "Creating Components") the use of component packages in Delphi applications. As I'm discussing the use of packages and DLLs for partitioning an application, here I'll start discussing the development of packages holding forms. We've seen earlier in this chapter that you can use forms inside DLLs, but this sometimes causes a few problems. If you are building both the library and the executable file in Delphi, using packages results in a much better and cleaner solution.

At first sight, you might believe that Delphi packages are a way to distribute components to be installed in the environment. Instead, you can use packages as a way to structure your code but, unlike DLLs, retain the full power of Delphi's OOP. Consider this: A package is a collection of compiled units and your program uses several units. The units the program refers to will be compiled inside the executable file, unless you ask Delphi to place them inside a package.

So how do you set up an application so that its code is split among one or more packages and a main executable file? You only need to compile some of the units in a package and then set up the options of the main program to dynamically link this package. For example, I've made a copy of the "usual" color selection form and renamed its unit as PackScrollF, then I've created a new package and added the unit to it, as you can see in Figure 12.7.

FIGURE 12.7:

The structure of the package hosting a form in Delphi's Package Editor



Files	Path
Contains	
PackScrollF	C:\md6code\12\PackForm
PackScrollF....	C:\md6code\12\PackForm
FormScroll	C:\md6code\12\PackForm
Requires	
vcl.dcp	

Before compiling this package, you should change its default output directories to refer to the current folder, not the standard `/Projects/Bpl` subfolder of Delphi. To do this, go to the Directories/Conditional page of the package Project Options, and set the current directory (a single dot, for short) for the Output directory (for the BPL) and DCP output directory. Then compile the package and do not install it in Delphi—there's no need to.

At this point, you can create a normal application and write the standard code you'll use in a program to show a secondary form, as in the following listing:

```
uses
  PackScrollF;

procedure TForm1.BtnChangeClick(Sender: TObject);
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  try
    // initialize the data
    FormScroll.SelectedColor := Color;
    // show the form
    if FormScroll.ShowModal = mrOK then
      Color := FormScroll.SelectedColor;
  finally
    FormScroll.Free;
  end;
end;
```

```
procedure TForm1.BtnSelectClick(Sender: TObject);
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  // initialize the data and UI
  FormScroll.SelectedColor := Color;
  FormScroll.BitBtn1.Caption := 'Apply';
  FormScroll.BitBtn1.OnClick := FormScroll.ApplyClick;
  FormScroll.BitBtn2.Kind := bkClose;
  // show the form
  FormScroll.Show;
end;
```

One of the advantages of this approach is that you can refer to a form compiled into a package with the exact same code you'll use for a form compiled in the program. In fact, if you simply compile this program, the unit of the form will actually be bound to it. To keep it in the package, you'll have to use run-time packages for the application and manually add the PackWithForm package to the list of run-time packages (this is not suggested by the Delphi IDE as we have not installed the package in the development environment).

Once you've done this step, compile the program and it will behave exactly as usual. But now the form is in a DLL package, and you can even modify the form in the package, recompile it, and simply run the application to see the effects. Notice, though, that for most changes affecting the interface portion of the units of the package (for example, adding a component or a method to the form), you should also recompile the executable program calling the package.

NOTE You can find the package and the program testing it in the `PackForm` folder of the source code related to the current chapter. The code of the next example is in the same folder.

Loading Packages at Run Time

In the example above, I indicated that the PackWithForm package is a run-time package to be used by the application. This means that the package is required to run the application and is loaded when the program starts. Both aspects can be avoided by loading the package dynamically, as we've done with DLLs. The resulting program will be more flexible, start more quickly, and use less memory.

An important element to keep in mind is that you'll need to call the `LoadPackage` and `UnloadPackage` Delphi functions rather than the `LoadLibrary` and `FreeLibrary` Windows API functions. The difference is that the functions provided by Delphi load the packages, but also call their proper initialization and finalization code.

Besides this important element—easy to accomplish once you know about it—the program will require some extra code, as we cannot refer from the main program to the unit hosting the form. We cannot use the form class directly, nor access to its properties or components. At least not with the standard Delphi code. Both issues, however, can be solved using class references, class registration, and RTTI (run-time type information). Let me start with the first one. In the form unit, in the package, I've added this initialization code:

```
initialization
  RegisterClass (TFormScroll);
```

As the package is loaded, the main program can use Delphi's `GetClass` function to get the class reference of the registered class and then call the `Create` constructor for this class reference.

To solve the second problem, I've made the `SelectedColor` property of the form in the package a published property, so that it is accessible via RTTI. Then I've replaced the code accessing this property (`FormScroll.Color`) with the following:

```
SetPropValue (FormScroll, 'SelectedColor', Color);
```

Summing up all of these changes, here is the code used by the main program (the DynaPackForm application) to show the modal form from the dynamically loaded package:

```
procedure TForm1.BtnChangeClick(Sender: TObject);
var
  FormScroll: TForm;
  FormClass: TFormClass;
  HandlePack: HModule;
begin
  // try to load the package
  HandlePack := LoadPackage ('PackWithForm.bpl');
  if HandlePack > 0 then
  begin
    FormClass := TFormClass(GetClass ('TFormScroll'));
    if Assigned (FormClass) then
    begin
      FormScroll := FormClass.Create (Application);
      try
        // initialize the data
        SetPropValue (FormScroll, 'SelectedColor', Color);
        // show the form
        if FormScroll.ShowModal = mrOK then
          Color := GetPropValue (FormScroll, 'SelectedColor');
      finally
        FormScroll.Free;
      end;
    end
    else
      ShowMessage ('Form class not found');
      UnloadPackage (HandlePack);
    end
    else
      ShowMessage ('Package not found');
  end;
```

Notice that the program unloads the package as soon as it is done with it. This is not compulsory. I could have moved the `UnloadPackage` call in the `OnDestroy` handler of the form, and avoided reloading the package after the first time.

Now you can try running this program without the package available, and you'll see that it will start properly, only to complain it cannot find the package as you click the Change button. In this program, you don't need to use run-time packages to keep the unit outside of your executable file, as you are not referring to the unit in your code. Also, the PackWithForm package doesn't need to be listed in the run-time packages. However, you must use run-time packages, or else your program will include the VCL global variables (as the `Application` object) and the dynamically loaded package will include another version, because it will refer to the VCL packages anyway.

Using Interfaces in Packages

Accessing the classes of the forms by means of methods and properties is much simpler than using RTTI all over the place. To build a larger application, I definitely try to use interfaces and to have multiple forms, each implementing a few standard interfaces defined by the program. An example cannot really do justice to this type of architecture, which becomes relevant for a large program, but I've tried nonetheless to build a program to show how this idea can be applied in practice.

NOTE If you don't know much about interfaces, I suggest you to refer to the related portion of Chapter 3 before reading this section.

To architect the IntfPack project, I've used three packages plus a demo application. Two of the three packages (called IntfFormPack and IntfFormPack2) define alternative forms used to select a color. The third package (called IntfPack) hosts a shared unit, used by both other packages. This unit basically includes the definition of the interface. I couldn't add it to both other packages because you cannot load two packages with a unit having the same name (even by run-time loading).

The only file of the IntfPack package is the IntfColSel unit, displayed in Listing 12.1. This unit defines the common interface (and you'll probably have a number of them in real-world application) plus a list of registered classes, which mimics Delphi's `RegisterClass` approach, but makes available the complete list so that you can easily scan it.

Listing 12.1: The IntfColSel unit of the IntfPack package

```
unit IntfColSel;

interface

uses
  Graphics, Contnrs;

type
  IColorSelect = interface
  ['{3F961395-71F6-4822-BD02-3B475FF516D4}']
    function Display (Modal: Boolean = True): Boolean;
    procedure SetSelColor (Col: TColor);
    function GetSelColor: TColor;
    property SelColor: TColor
      read GetSelColor write SetSelColor;
  end;

procedure RegisterColorSelect (AClass: TClass);

var
  ClassesColorSelect: TClassList;

implementation

procedure RegisterColorSelect (AClass: TClass);
begin
  if ClassesColorSelect.IndexOf (AClass) < 0 then
    ClassesColorSelect.Add (AClass);
end;

initialization
  ClassesColorSelect := TClassList.Create;

finalization
  ClassesColorSelect.Free;

end.
```

Once we have this interface available, we can define forms that implement it, as in the following example, taken form the IntfFormPack:

```
type
  TFormSimpleColor = class(TForm, IColorSelect)
    ...
  private
    procedure SetSelColor (Col: TColor);
    function GetSelColor: TColor;
  public
    function Display (Modal: Boolean = True): Boolean;
```

The two access methods simply read and write the value of the color from some components of the form (a simple ColorGrid in this specific case), while the `Display` method internally calls either `Show` or `ShoModal`, depending on the parameter:

```
function TFormSimpleColor.Display(Modal: Boolean): Boolean;
begin
  Result := True; // default
  if Modal then
    Result := (ShowModal = mrOK)
  else
  begin
    BitBtn1.Caption := 'Apply';
    BitBtn1.OnClick := ApplyClick;
    BitBtn2.Kind := bkClose;
    Show;
  end;
end;
```

The form is structured like that of the last example, still available in the second package, and has an OK button that is turned into an Apply button. Finally, the unit has the registration code in the `initialization` section, so that it is executed when the package is dynamically loaded:

```
RegisterColorSelect (TFormSimpleColor);
```

With this architecture in place, we can build a rather elegant and flexible main program, which is based on a single form. When the form is created, it defines a list of packages (called `HandlesPackages`) and loads them all. I've hard-coded the package in the code of the example, but of course you can as well search for the packages of the current folder or use a configuration file to make the application structure more flexible. Finally, after loading the packages, the program shows the registered classes in a list box. This is the code of the `LoadDynaPackage` and `FormCreate` methods:

```
procedure TFormUseIntf.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  // loads all runtime packages
  HandlesPackages := TList.Create;
  LoadDynaPackage ('IntfFormPack.bpl');
  LoadDynaPackage ('IntfFormPack2.bpl');

  // add class names and select the first
  for I := 0 to ClassesColorSelect.Count - 1 do
    lbClasses.Items.Add (ClassesColorSelect [I].ClassName);
  lbClasses.ItemIndex := 0;
end;
```

```
procedure TFormUseIntf.LoadDynaPackage(PackageName: string);
var
  Handle: HModule;
begin
  // try to load the package
  Handle := LoadPackage (PackageName);
  if Handle > 0 then
    // add to the list for later removal
    HandlesPackages.Add (Pointer(Handle))
  else
    ShowMessage ('Package ' + PackageName + ' not found');
end;
```

The main reason for keeping the list of package handles is to be able to unload them all when the program ends. In fact, we don't need these handles to access the forms defined in those packages. The run-time code used to create and show a form simply uses the corresponding component classes. This is a snippet of code used to display a modeless form (an option controlled by a check box):

```
var
  AComponent: TComponent;
  ColorSelect: IColorSelect;
begin
  AComponent := TComponentClass
    (ClassesColorSelect[LbClasses.ItemIndex]).Create (Application);
  ColorSelect := AComponent as IColorSelect;
  ColorSelect.SelColor := Color;
  ColorSelect.Display (False);
```

The program actually uses the `Supports` function to check that the form really does support the interface before using it, and also accounts for the modal version of the form, but its essence is properly depicted in the four statements above. By the way, notice that the code doesn't actually require a form. A nice exercise would be to add to the architecture a package with a component encapsulating the color selection dialog box or inheriting from it.

WARNING The main program refers to the unit hosting the interface definition but should not link this file in. Rather, it should use the run-time package containing this unit, as the dynamically loaded packages do. Otherwise the main program will use a different copy of the same code, including a different list of global classes. It is this list of global classes, not the use of the same interface, that should not be duplicated in memory.

Packages Versus DLLs

In the preceding section, we've seen that using packages is a fine alternative to using DLLs for sharing compiled code among multiple Delphi applications or splitting a large executable into multiple (and partially independent) modules. As a summary, here are a few of the differences between the two approaches:

- DLLs are collections of functions; packages can easily "export" classes and objects.
- Dynamically loading a DLL implies losing any safety in the function call, in case you pass the wrong parameters. Dynamically loading packages requires some extra coding as well, but is definitely simpler and safer, particularly if you use interfaces.
- Packages force you to use the VCL run-time package for the application, although even when using DLLs, run-time packages help solve quite a few difficulties (as we'll discuss shortly).
- DLLs can be used across programming languages and development environments, but packages are limited to Delphi and C++Builder. If you need libraries in a Delphi-only environment, packages are the native solution and should generally be preferred.

Using DLLs also accounts for a few extra troubles that you can partially solve by letting the DLLs share run-time packages. The following sections discuss the problem briefly, as this is not the recommended approach anyway.

Executables and DLLs Sharing the VCL Packages

In the FormDLL example, we faced a problem: When you place forms inside a DLL, you don't get the proper behavior for the flat buttons even if you synchronize the two application objects. Moreover, both the executable file and the DLL contain the compiled code of the VCL library, leading to useless duplication. As discussed earlier, the simplest solution to this issue is to use a package instead of a DLL.

Another solution is to keep the DLL in its format, but let it use run-time packages, so that no global objects will be duplicated between the executable and the library. In this case there will be only one `Application` object, shared by the program and the DLL, instead of two separate objects, so we don't need the synchronization code any more.

Another simplification to the program comes from the fact that the modeless form inside the DLL can communicate back to the main form by accessing the list of the forms (available to the shared global `Screen` object) or simply using the `Application.MainForm` property. This is what I've done in the FormDllP example on the CD.

With this approach, you face the risk of having the main form and the form in the DLL not synchronized at all, with two entries in the Taskbar; also, this code still has all the other problems of the first version of the FormDLL example. The problem lies in the fact that when you run the program, the DLL is initialized before the application, so it is the DLL that initializes the Forms unit of the VCL. Within a DLL, the VCL creates the Application object but doesn't create the corresponding window.

There are two radically different approaches to this initialization issue: One is to change the initialization order by loading the DLL dynamically after the application has started; the second is to add some extra initialization code in the program. None of these techniques provides a better solution than using packages altogether!

Dynamically Loading the DLL with Packages

The first solution is demonstrated by the FirstDLLD library and the UseDyna example, which dynamically loads the DLL built with run-time packages. The main program loads the DLL at startup, in the OnCreate event handler of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  hInstDll := SafeLoadLibrary ('FormDllD.dll');
  if hInstDll <= 0 then
    raise Exception.Create ('FormDllD library not found');
end;
```

In the program I haven't declared the functions exported by the DLL, to avoid the implicit link of the library. Instead I've declared two procedure types:

```
type
  TGetColorProc = function (Col: LongInt): LongInt; stdcall;
  TShowColorProc = procedure (Col: LongInt); stdcall;
```

These types are used for converting the generic pointer returned by the GetProcAddress function, as we've already seen in the DynaCall example:

```
procedure TForm1.BtnChangeClick(Sender: TObject);
var
  Col: LongInt;
  GetColorProc: TGetColorProc;
  FPointer: TFarProc;
begin
  FPointer := GetProcAddress (hInstDll, 'GetColor');
  if FPointer = nil then
    raise Exception.Create ('GetColor DLL function not found');
  GetColorProc := TGetColorProc (FPointer);
  // original code
  Col := ColorToRGB (Color);
  Color := GetColorProc (Col);
end;
```

Using dynamic loading is the correct approach, officially supported by Delphi. Still, you have to call the functions dynamically, which requires a little extra coding.

Fixing the Initialization Code

An alternate solution is to keep the external functions defined in the main program, let the DLL start first and initialize the VCL, and let the VCL create the `Application` object without the connected window. In fact, we can add one line of code to the library to ask for the creation of the window of the `Application` object during the library initialization process (before the executable creates its own main objects). We accomplish this by writing the code in the initialization section of one of the units of the DLL:

```
initialization
  Application.CreateHandle;
```

Because this code is in the DLL, the application fails to load its icon. The solution is actually very simple. In the `OnCreate` event handler of the main form (in the main program), simply reload the current icon:

```
Application.Icon.Handle := LoadIcon (HInstance, 'MAINICON');
```

Exploring the Structure of a Package

You might wonder: is it possible to know whether a unit has been linked in the executable file or if it's part of a run-time package? Not only is this possible in Delphi, but you can also explore the overall structure of an application. A component can use the undocumented `ModuleIsPackage` global variable, declared in the SysInit unit. You should never need this, but it is technically possible for a component to have different code depending on whether it is packaged or not. The following code extracts the name of the run-time package hosting the component, if any:

```
var
  fPackName: string;
begin
  // get package name
  SetLength (fPackName, 100);
  if ModuleIsPackage then
  begin
    GetModuleFileName (HInstance, PChar (fPackName), Length (fPackName));
    fPackName := PChar (fPackName) // string length fixup
  end
  else
    fPackName := 'Not packaged';
```

Besides accessing package information from within a component (as in the code above), you can also do so from a special entry point of the package libraries, the `GetPackageInfoTable` function. This function returns some specific package information that Delphi stores as resources and includes in the package DLL. Fortunately, we don't need to use low-level techniques to access this information, since Delphi provides some high-level functions to manipulate it.

You can use two functions to access package information:

- `GetPackageDescription` returns a string that contains a description of the package. To call this function, you must supply the name of the module (the package library) as the only parameter.
- `GetPackageInfo` doesn't directly return information about the package. Instead, you pass it a function that it calls for every entry in the package's internal data structure. In practice, `GetPackageInfo` will call your function for every one of the package's contained units and required packages. In addition, `GetPackageInfo` sets several flags in an Integer variable.

These two function calls allow us to access internal information about a package, but how do we know which packages our application is using? You could determine this by looking at an executable file using low-level functions, but Delphi helps you again by supplying a simpler approach. The `EnumModules` function doesn't directly return information about an application's modules but allows you to pass it a function, which it calls for each module of the application, the main executable file, and for each of the packages the application relies on.

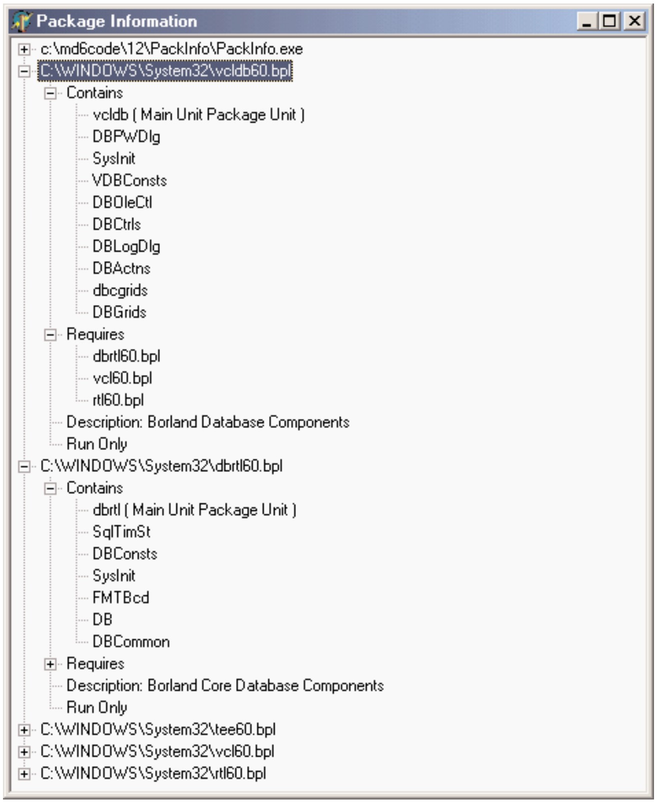
To demonstrate this approach, I've built a simple example program that displays the module and package information in a TreeView component. Each first-level node corresponds to a module, and within each module I've built a subtree that displays the contained and required packages for that module, as well as the package description and compiler flags (`RunOnly` and `DesignOnly`). You can see the output of this example in Figure 12.8.

In addition to the TreeView component, I've added several other components to the main form, but hidden them from view: a DBEdit, a Chart, and a FilterComboBox. I added these components simply to include more run-time packages in the application, beyond the ubiquitous `VCL60.BPL`. The only method of the form class is `FormCreate`, which calls the module enumeration function:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  EnumModules(ForEachModule, nil);
end;
```

FIGURE 12.8:

The output of the PackInfo example, with the details of the packages it uses



```
⊞ c:\md6code\12\PackInfo\PackInfo.exe
⊟ C:\WINDOWS\System32\vcldb60.bpl
    ⊟ Contains
        vcldb ( Main Unit Package Unit )
        DBPWDlg
        SysInit
        VDBConsts
        DBOleCtl
        DBCtrls
        DBLogDlg
        DBActns
        dbcgrids
        DBGrids
    ⊟ Requires
        dbrtl60.bpl
        vcl60.bpl
        rtl60.bpl
    Description: Borland Database Components
    Run Only
⊟ C:\WINDOWS\System32\dbrtl60.bpl
    ⊟ Contains
        dbrtl ( Main Unit Package Unit )
        SqlTimSt
        DBConsts
        SysInit
        FMTBcd
        DB
        DBCommon
    ⊞ Requires
    Description: Borland Core Database Components
    Run Only
⊞ C:\WINDOWS\System32\tee60.bpl
⊞ C:\WINDOWS\System32\vcl60.bpl
⊞ C:\WINDOWS\System32\rtl60.bpl
```

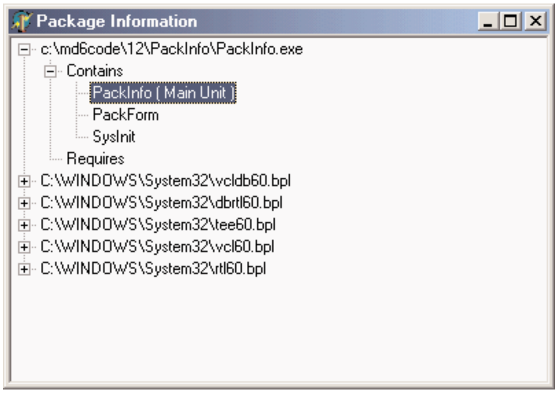
The EnumModules function accepts two parameters. The first is the callback function (in our case, ForEachModule), and the second is a pointer to a data structure that the callback function will use (in our case, nil, since we didn't need this). The callback function must accept two parameters—an HInstance value and an untyped pointer—and must return a Boolean value. The EnumModules function will, in turn, call our callback function for each module, passing the instance handle of each module as the first parameter and the data structure pointer (nil in our example) as the second.

```
function ForEachModule (HInstance: Longint;
  Data: Pointer): Boolean;
var
  Flags: Integer;
```

```
    ModuleName, ModuleDesc: string;
    ModuleNode: TTreeNode;
  begin
    with Form1.TreeView1.Items do
    begin
      SetLength (ModuleName, 200);
      GetModuleFileName (HInstance,
        PChar (ModuleName), Length (ModuleName));
      ModuleName := PChar (ModuleName); // fixup
      ModuleNode := Add (nil, ModuleName);

      // get description and add fixed nodes
      ModuleDesc := GetPackageDescription (PChar (ModuleName));
      ContNode := AddChild (ModuleNode, 'Contains');
      ReqNode := AddChild (ModuleNode, 'Requires');

      // add information if the module is a package
      GetPackageInfo (HInstance, nil, Flags, ShowInfoProc);
      if ModuleDesc <> '' then
      begin
        AddChild (ModuleNode, 'Description: ' + ModuleDesc);
        if Flags and pfDesignOnly = pfDesignOnly then
          AddChild (ModuleNode, 'Design Only');
        if Flags and pfRunOnly = pfRunOnly then
          AddChild (ModuleNode, 'Run Only');
      end;
    end;
    Result := True;
  end;
```

As you can see in the preceding code, the ForEachModule function begins by adding the module name as the main node of the tree (by calling the Add method of the TreeView1.Items object and passing nil as the first parameter). It then adds two fixed child nodes, which are stored in the ContNode and ReqNode variables declared in the implementation section of this unit.

Next, the program calls the GetPackageInfo function and passes it another callback function, ShowInfoProc, which I'll discuss shortly. The program adds the details for the main module (see Figure 12.9), simply because this will provide a list of the application's units. At the end of this function, we add more information if the module is a package, such as its description and compiler flags (we know it's a package if its description isn't an empty string).



Package Information

```
c:\md6code\12\PackInfo\PackInfo.exe
  Contains
    PackInfo ( Main Unit )
    PackForm
    SysInit
  Requires
C:\WINDOWS\System32\vcldb60.bpl
C:\WINDOWS\System32\dbrtl60.bpl
C:\WINDOWS\System32\tee60.bpl
C:\WINDOWS\System32\vcl60.bpl
C:\WINDOWS\System32\rtl60.bpl
```

FIGURE 12.9: The PackInfo example also lists the units that are part of the current application.

Earlier, I mentioned passing another callback function, the ShowInfoProc procedure, to the GetPackageInfo function, which in turn calls our callback function for each contained or required package of a module. This procedure creates a string that describes the package and its main flags (added within parentheses), and then inserts that string under one of the two nodes (ContNode and ReqNode), depending on the type of the module. We can determine the module type by examining the NameType parameter. Here is the complete code of our second callback function:

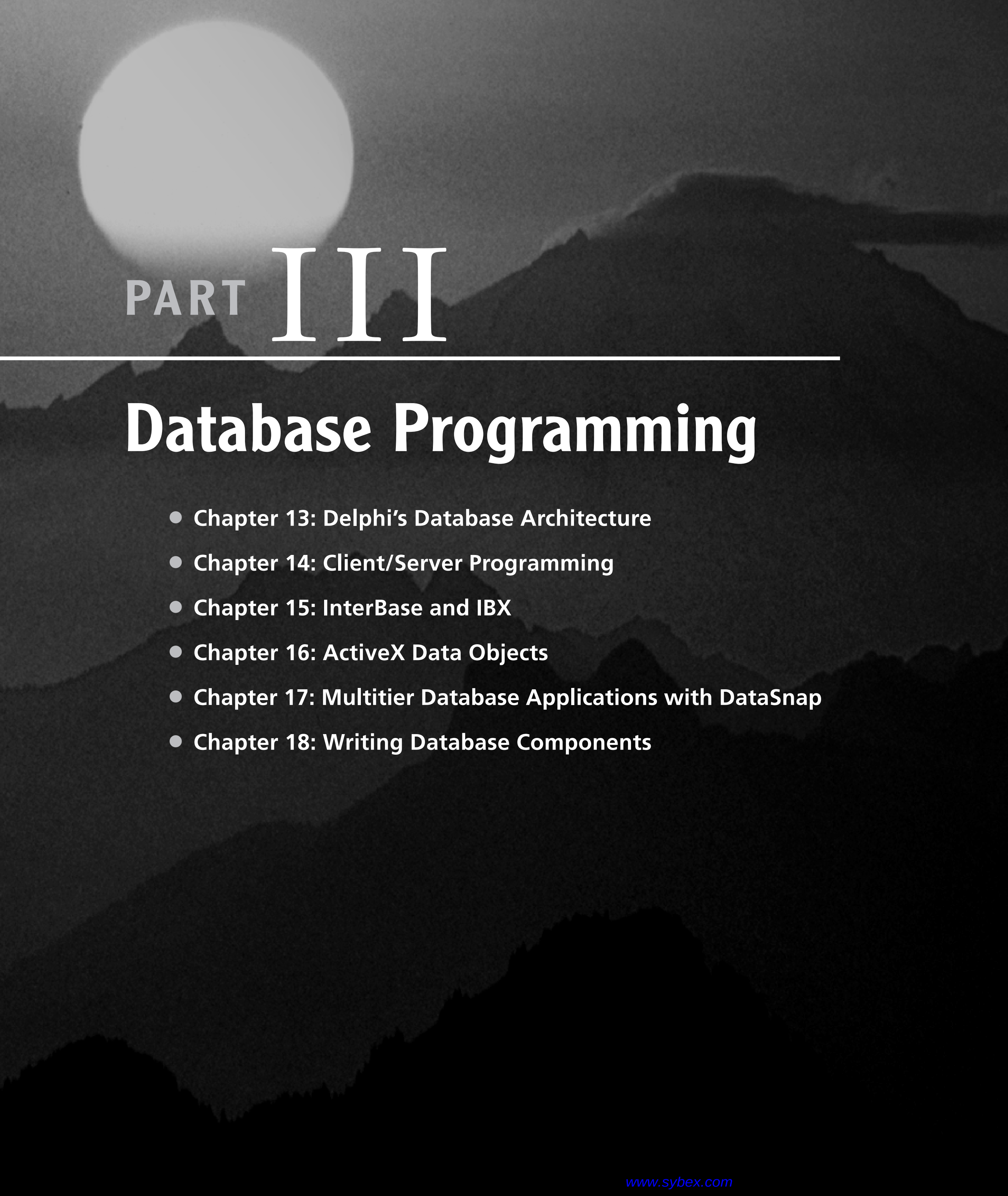
```
procedure ShowInfoProc (const Name: string; NameType: TNameType; Flags: Byte;
  Param: Pointer);
var
  FlagStr: string;
begin
  FlagStr := ' ';
  if Flags and ufMainUnit <> 0 then
    FlagStr := FlagStr + 'Main Unit ';
  if Flags and ufPackageUnit <> 0 then
    FlagStr := FlagStr + 'Package Unit ';
  if Flags and ufWeakUnit <> 0 then
    FlagStr := FlagStr + 'Weak Unit ';
  if FlagStr <> ' ' then
    FlagStr := ' (' + FlagStr + ')';
  with Form1.TreeView1.Items do
    case NameType of
      ntContainsUnit: AddChild (ContNode, Name + FlagStr);
      ntRequiresPackage: AddChild (ReqNode, Name);
    end;
end;
```

Here, you'll notice that the Flags parameter doesn't contain flag style information, as the online help seems to imply. If you want to investigate this topic further, examine the SysUtils unit.

What's Next?

In this chapter we have seen how you can call functions that reside in DLLs, how to create DLLs using Delphi, and how to use strings and place Delphi forms inside a library. Another technique for placing Delphi forms and other classes in libraries is to use packages, special DLLs that the IDE uses for installing components, but that you can use for dividing an application into multiple executable files.

I will get back to the topics of libraries that expose objects and classes when I discuss COM and OLE in Chapters 19 and 20. For the moment, instead, we'll move to a totally different topic, the development of database-oriented and client/server applications with Delphi.



PART III

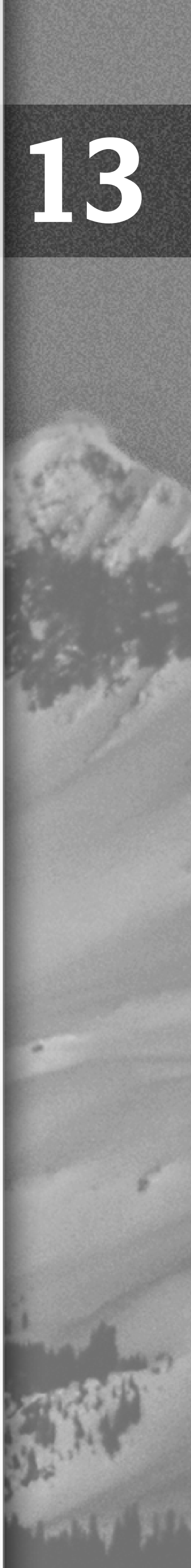
Database Programming

- Chapter 13: Delphi’s Database Architecture
- Chapter 14: Client/Server Programming
- Chapter 15: InterBase and IBX
- Chapter 16: ActiveX Data Objects
- Chapter 17: Multitier Database Applications with DataSnap
- Chapter 18: Writing Database Components

CHAPTER 13

Delphi's Database Architecture

- Delphi's database components
- Database access alternatives
- Using data-aware controls
- The DBGrid and multirecord objects
- Manipulating table fields
- Database applications with standard controls



Delphi's support for database applications is one of the key features of the programming environment. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming.

What you won't find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports database access.

I'll begin with an explanation of the alternatives Delphi offers in terms of data access, and then I'll provide an overview of the database components that are available in Delphi. This chapter includes an overview of the `TDataSet` class, an in-depth analysis of the `TField` components, and the use of data-aware controls. The following chapters will provide information on more advanced database programming topics, such as client/server programming, the use of dbGo, dbExpress, and InterBase Express.

Accessing a Database: BDE, dbExpress, and Other Alternatives

In the first few versions of Delphi, the only available technology to access database data was to use the Borland Database Engine (BDE). Starting with Delphi 3, the portion of VCL related to Database access has been restructured to open it up to multiple database access solutions. Delphi 5 saw the introduction of specific sets of components supporting Microsoft's ActiveX Data Objects (ADO) and InterBase Express (IBX). Delphi 6 adds to the picture dbExpress, which is a brand-new cross-platform and database-independent data-access technology provided by Borland with Kylix on Linux and Delphi 6 on Windows.

With all these alternatives, it is easy to get confused on which approach to use. In the following sections I've provided a short description of the key elements of these data-access technologies available in Delphi, trying to suggest in which case you'll want to use each of them.

Borland Database Engine (BDE)

The BDE originated with Paradox, well before Delphi existed, and was extended by Borland to support other local databases and many SQL servers. The BDE has direct access to dBASE, Paradox, ASCII, FoxPro, and Access tables. A series of drivers (called SQL Links and available only in Delphi Enterprise) allows access to some SQL servers, including Oracle, Sybase, Microsoft, Informix, InterBase, and DB2 servers. If you need access to a different database, the BDE can also interface with ODBC drivers.

The advantage of using a common database engine is that your application will be portable among different servers of the same category (porting from a local database to an SQL server is generally much more complex). The specific advantages of using the BDE are that this technology is very well integrated in Delphi; its elements are very well documented; and it is the only viable solution for accessing local files such as Paradox and dBase tables.

The disadvantages of this solution: Borland has stopped developing it (there will be no further updates); you'll have to install and configure it on the client computers; it is quite a "heavyweight" engine, with large installation files and memory requirements; and it is available only on Windows. If you have existing Delphi BDE applications accessing local files, there is no hurry to convert them and get rid of the BDE, unless you want to move your applications to Linux. If you are using an SQL server, migrating to another data-access technology will probably be easier.

BDE is still a good solution, if you balance advantages and disadvantages, but its long-term viability is certainly in doubt. I'll keep using the BDE for the simpler examples of this chapter, but only for the sake of simplicity. In any case, I'll try to stress the elements common to all the dataset components rather than focus on specific features of BDE or Paradox.

The Delphi components related to the BDE are all hosted in the Data Access page of the Components palette. There are three dataset components, Table, Query, and StoredProc, plus the UpdateSQL used in connection with the Query component. The Database and Session components are used to set up the database connection. The BatchMove component is for copying data; the rarely used NestedTable component allows you to nest master-detail data in a sub-table; and the BDEClientDataSet component, introduced in Delphi 6, merges a ClientDataSet with a BDE-related data-access component.

ActiveX Data Objects (ADO)

ADO, which stands for ActiveX Data Objects, is Microsoft's high-level interface for database access. ADO is implemented on Microsoft's data-access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. ADO is an engine with features comparable to the BDE: database server independence supporting local and SQL servers alike, a really heavyweight engine, and a simplified configuration (because it is not centralized). Installation should in theory not be an issue, as the engine is part of recent versions of Windows. However, the limited compatibility among versions of ADO will force you to upgrade your users' computers to the same version you've used for developing the program—and the sheer size of the MDAC (Microsoft Data Access Components) installation, which updates large portions of the operating system, makes this operation far from simple.

ADO offers some definite advantages if you plan on using Access or SQL Server, as Microsoft's drivers for their own databases are of better quality than the average OLE DB providers. For Access databases, specifically, using Delphi's ADO components is a good solution. But if you plan using other SQL servers, first check the availability of a good quality driver, as you might have some surprises. ADO is very powerful, but you have to learn living with it, as it really stands in the way between your program and the database, providing services but occasionally also issuing different commands than you are expecting. On the negative side, do not even think of using ADO if you plan future cross-platform development: this Microsoft-specific technology is not available on Linux or other operating systems.

In short, use ADO if you plan working only on Windows, want to use Access or other Microsoft databases, or you find a good OLE DB provider for each of the database servers you plan working with (at the moment, for example, this excludes InterBase and many other SQL servers).

ADO components (part of a package Borland called ADO Express in Delphi 5 and now calls dbGo in Delphi 6) are all grouped in the ADO page of the Components palette. The three core components are ADOConnection (for database connection), ADOCommand (for executing SQL commands), and ADODataSet (for executing requests that return a result set). There are also three compatibility components—ADOTable, ADOQuery, and ADOStoredProc—which you can use for porting BDE-based applications to ADO. Finally, there is the RDSConnection component, for accessing data in remote multitier applications.

NOTE Chapter 16, "ActiveX Data Objects," covers ADO and related technologies in great detail.

The dbExpress Library

One of the relevant new features of Delphi 6 is the introduction of the dbExpress database library for the Windows platform. I say "library" because, unlike BDE and ADO, dbExpress uses a lightweight approach; and I underline "Windows" because the same library is available also for Linux in Borland Kylix.

Being light and portable are actually the two key characteristics of dbExpress and the reasons it has been introduced by Borland, along with the development of the Kylix project. There are certainly other database libraries you could use in the past and can still use with Delphi, but this new offering is worth a thought. Consider also it requires basically no configuration on the user machines.

Compared to other powerhouses, dbExpress is really limited in its capabilities. It can access only SQL servers (no local files); it has no caching capabilities and provides only unidirectional access to the data; it can natively work only with SQL queries and is unable of generating the corresponding SQL update statements.

At first sight, you might think that these limitations make the library pretty useless. On the contrary, these are *features* that make it interesting. Unidirectional datasets with no direct update are the norm if you need to produce reporting, including generating HTML pages showing the content of a database. If you want to build a user interface to edit the data, instead, consider that Delphi includes specific components (the ClientDataSet and Provider, in particular) that provide caching and query resolution. These components allow your dbExpress-based application to have much more control than you can have with a separate (and monolithic) database engine, which does extra things for you but often does it the way it wants, not the way you would like.

Considering that Borland is pushing this library, and it is the only viable database-independent solution on Linux, I really urge you to consider it for new applications, and even to think about updating existing Delphi applications to this new architecture.

InterBase Express (IBX)

Delphi includes components for native access to Borland's own open-source (and free) InterBase server. Unlike BDE, ADO, and dbExpress, this is not a server-independent database engine, but a technology for accessing a specific database server. If you plan using only InterBase as your back-end RDBMS, using a specific set of components can give you more control of the server, provide the best performance, and allow you also to configure and maintain the server from within a custom client application.

NOTE The use of InterBase Express highlights the case of database-specific custom datasets, which are available from third-party vendors for many servers (there are other dataset components for InterBase, as there are for Oracle, Access, dBase files, and many others).

In short, you can consider using InterBase Express (or other comparable sets of components) if you are sure you won't change your database and want to achieve best performance and control at the expense of flexibility and portability. The down side is that the extra performance and control you gain might be limited, and you'll have to learn how to use another set of components with a specific behavior, compared to learning how to use a generic engine and applying your knowledge to different situations.

The ClientDataSet Component

Finally, there is a component derived from `TDataSet` that has a peculiar behavior and can be combined with other data-access components. The ClientDataSet component, in fact, is a dataset accessing data kept in memory. The in-memory data can be totally temporary (lost as you exit the program), saved to a local file as a snapshot, and imported by another dataset using a Provider component. This last situation is certainly the most common: You can hook a ClientDataSet to any other local dataset, or use Borland's multitier support (discussed in

Chapter 17, "Multitier Database Applications with DataSnap") to retrieve data from a dataset hosted by a different application, possibly running on a separate computer.

The ClientDataSet component becomes particularly useful if the data-access components you are using provide limited or no caching. This is particularly true of the new dbExpress engine, but can equally help you when using the BDE or other native components. On the other hand, ADO already provides most of the services of the ClientDataSet component and using these two at the same time can be useful only in limited situations.

Classic BDE Components

Each of the database-access solutions discussed above has its own set of data-access, database connection, and extra utility components on a specific page of the Component palette. In Delphi 6, the classic BDE components have been moved to the new BDE page and include the Table, Query, and StoredProc components. The ADO, dbExpress, and InterBase Express components are each in specific pages, and all include specific dataset components and others that tend to mimic the BDE components, simplifying the porting of existing applications. The Data Access page of the Component palette in Delphi 6 includes only the Data Source component and others not specifically related with any single data access technology.

Besides the data-access component of your choice, a Delphi visual application generally uses some data-aware controls (in the Data Controls page) and the DataSource component. Data-aware controls are visual components used to view and edit the data in a form and are extensions of standard components such as edit and list boxes, radio buttons, images, and the grid. The DataSource component has the role of connector between the data-aware controls and a dataset component.

Tables and Queries

The simplest traditional way to specify data access in Delphi was to use the BDE Table component. A Table object simply refers to a database table. When you use a Table component, you need to indicate the name of the database you want to use in its `DatabaseName` property. You can enter an alias or the path of the directory with the table files. The Object Inspector lists the available names, which depend on the aliases installed in the BDE.

You also need to indicate a proper value in the `TableName` property. The Object Inspector lists the available tables of the current database (or directory), so you should generally select the `DatabaseName` property first.

Another classic dataset is the BDE Query component. A query requires a SQL language command. You can customize a query using SQL more easily than you can customize a table (as long as you know at least the basic elements of SQL, of course). The Query component has a DatabaseName property like the Table component, but it does not have a TableName property. The table is indicated in the SQL statement, stored in the SQL property.

For example, you can write a simple SQL statement like this:

```
select * from Country
```

where Country is the name of a table and the asterisk (*) indicates that you want to use all of the fields in the table.

The efficiency of a table or a query varies depending on the database you are using. In general, we can say that the Table component tends to be faster on local tables, while the Query component tends to be faster on SQL servers, although this is just a very general rule, and in many cases you might have the opposite effect. We'll see some efficiency issues while discussing client/server development in Chapter 14, "Client/Server Programming."

The third BDE dataset component is StoredProc, which refers to stored procedures of a SQL server database. You can run these procedures and get the results in the form of a database table. Stored procedures can only be used with SQL servers.

Specific Table Features

The BDE Table component has specific features not shared by all datasets. For example, it has filters, ranges, and specific techniques for locating records. A filter, set in the Filter property and activated by toggling the Filtered property, is available in each dataset, although its role changes depending on the underlying implementation. A range, instead, is specific to a Table and allows you to specify the two extreme values and consider only the record falling within that interval.

When using a Table, and particularly a local one, there are specific methods you can use to find a record, such as GotoKey, FindKey, GotoNearest, FindNearest, and Locate. The Locate method is shared by all datasets, and I'll discuss it later along with other general features of the TDataSet class. The other methods are specific of the TTable class and work in conjunction with the index set in the ndexFieldNames property of the component.

The simplest approach is to use the FindNearest method for the approximate search and the FindKey method to look for an exact match:

```
// goto
Table1.FindNearest ([EditName.Text]);

// go near
if not Table1.FindKey ([EditName.Text]) then
  MessageDlg ('Name not found', mtError, [mbOk], 0);
```

Both find methods use as parameters an array of constants. Each array element corresponds to one of the fields of the current index. You can also pass only the value for the initial field or fields of the index, so the following fields will not be considered.

NOTE I won't discuss these features in details, showing complete examples, because some of them are limited to the BDE and makes sense only for local tables, not for SQL server–based tables. Actually if you set a filter or a range over a Table connected with a SQL server, the BDE will try to generate a proper select statement, avoiding fetching all the data and filtering it locally. The problem is that this isn't always possible and you lose most of your control, two good reasons to use the Query component when working with SQL servers.

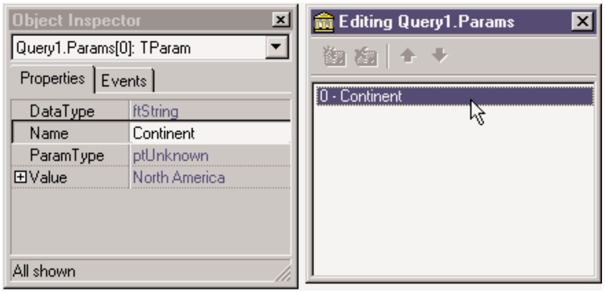
A Query with Parameters

When you need slightly different versions of the same SQL query, instead of modifying the text of the Query (stored in the `SQL` property) each time, you can write a query with a parameter and simply change the value of the parameter. For example, if you decide to have a user choose the countries of a continent (using the Country table of the DBDEMOS database), you can write the following parametric query:

```
select *
from Country
where Continent = :Continent
```

In this SQL clause, `:Continent` is a parameter. We can set its data type and startup value, using the editor of the `Params` property collection of the Query component. When the Parameters collection editor is open, as shown in Figure 13.1, you see a list of the parameters defined in the SQL statement and set the data type and the initial value of these parameters.

FIGURE 13.1: Editing the collection of parameters of a Query component


Object Inspector

Query1.Params[0]: TParam

Properties	Events
DataType	ftString
Name	Continent
ParamType	ptUnknown
⊞Value	North America

All shown

Editing Query1.Params

0 - Continent

The form displayed by this program, called ParQuery and available on the companion CD, uses a list box to provide all the available values for the parameters. Instead of preparing the items of the list box at design time, we can extract the available continents from the same database table as the program starts. This is accomplished using a second query component, with this SQL statement:

```
select distinct Continent
from Country
```

After activating this query, the program scans its result set, extracting all the values and adding them to the list box:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
  // get the list of continents
  Query2.Open;
  while not Query2.EOF do
  begin
    ListBox1.Items.Add (Query2.Fields [0].AsString);
    Query2.Next;
  end;
  ListBox1.ItemIndex := 0;

  // open the first query
  Query1.Params[0].Value := ListBox1.Items [0];
  Query1.Open;
end;
```

Before opening the query, the program selects as its parameter the first item of the list box, which is also activated by setting the ItemIndex property to 0. When the list box is selected, the program closes the query and changes the parameter:

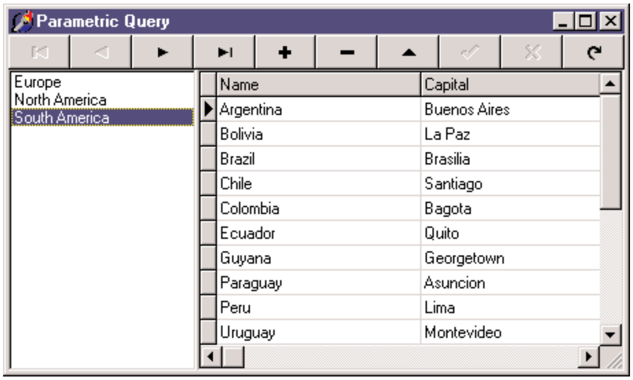
```
procedure TQueryForm.ListBox1Click(Sender: TObject);
begin
  Query1.Close;
  Query1.Params[0].Value := ListBox1.Items [Listbox1.ItemIndex];
  Query1.Open;
end;
```

This displays the countries of the selected continent in the list box, as you can see in Figure 13.2. The final refinement is that when the user enters a record with a new continent, it is added automatically to the list box. Instead of refreshing the entire list, with the same code executed in the FormCreate method, we can do this by handling the BeforePost event and adding the continent to the list if it is not already there:

```
procedure TQueryForm.Query1BeforePost(DataSet: TDataSet);
var
  StrNewCont: string;
begin
  // add the continent, if not already in the list
  StrNewCont := Query1.FieldByName ('Continent').AsString;
  if ListBox1.Items.IndexOf (StrNewCont) < 0 then
    ListBox1.Items.Add (StrNewCont);
end;
```

FIGURE 13.2:

The ParQuery example at run time



Europe
North America
South America

Name	Capital
Argentina	Buenos Aires
Bolivia	La Paz
Brazil	Brasilia
Chile	Santiago
Colombia	Bagota
Ecuador	Quito
Guyana	Georgetown
Paraguay	Asuncion
Peru	Lima
Uruguay	Montevideo

We can add a little extra code to this program to take advantage of a specific feature of parameterized queries. To react faster to a change in the parameters, these queries can be optimized, or *prepared*. Simply call the `Prepare` method before the program first opens the query (after setting the `Active` property of the Query component to `False` at design time) and call `Unprepare` once the query won't be used anymore:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
  ...
  // prepare and open the first query
  Query1.Prepare;
  Query1.Params[0].Value := ListBox1.Items [0];
  Query1.Open;
end;

procedure TQueryForm.FormDestroy(Sender: TObject);
begin
  Query1.Close;
  Query1.Unprepare;
end;
```

Prepared parameterized queries are very important when you work on a complex query. In fact, the BDE or the SQL server must read the text of the query and determine how to process it. If you use the same query (even if a parametric one) over and over, the engine doesn't need to reprocess the query but already knows how to handle it.

Master/Detail Structures

Often you need to relate tables, which have a one-to-many relationship. This means that for a single record of the master table, there are many detailed records in a secondary table. A classic example is that of an invoice and the items of the invoice; another is a list of customers and the orders each customer has made. This is very common situation in database programming, and Delphi provides explicit support for it with the master/detail structure. We'll see this structure for BDE Table and Query components, but the same technique applies to almost all of the datasets available in Delphi.

NOTE The `TDataSet` class has a generic `DataSource` property for setting up a master data source, but the Table component, for example, uses a different property (`MasterSource`) to express the same concept.

Master/Detail with Tables

The simplest ways to create a master/detail structure in Delphi is to use the Database Form Wizard, selecting a master/detail form in the first page. To accomplish the same effect manually, place two table components in a form or data module, connect them with the same database, and connect each with a table. In the MastDet example, I've used the customer and orders tables of the DBDEMOS database, and I've used a data module. Now add a DataSource component for each table, and for the secondary table set a master source to the data source connected to the first table. Finally relate the secondary table to a field (called `MasterField`) of the main table, using the special property editor provided.

A Data Module for Data-Access Components

To build a Delphi database application, you can place data-access components and the data-aware controls in a form. This is handy for a simple program, but having the user interface and the data access and data model in a single, often large, unit is far from a good idea. For this reason, Delphi implements the idea of *data module*, a container of nonvisual components I already introduced in Chapter 1, "The Delphi 6 IDE."

At design time, a data module is similar to a form, but at run time it exists only in memory. The `TDataModule` class derives directly from `TComponent`, so it is completely unrelated to the Windows concept of a window (and is fully portable among different operating systems). Unlike a form, a data module has just a few properties and events. For this reason, it's useful to think of data modules as components and method containers.

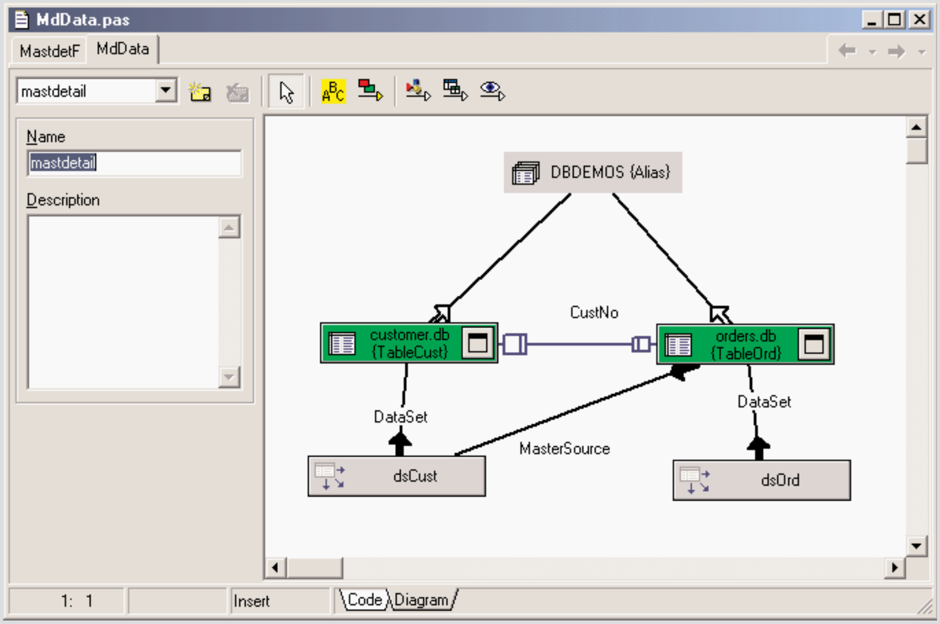
Like a form or a frame, a data module has a designer. This means Delphi creates for a data module a specific Object Pascal unit for the definition of its class and a form definition file that lists its components and their properties.

There are several reasons to use data modules. The simplest one is to share data-access components among multiple forms, as I'll demonstrate at the beginning of the next chapter. This technique works in conjunction with visual form linking, the ability to access components of another form or data module at design time (with the File ➢Use Unit command). The second

Continued on next page

reason is to separate the data from the user interface, improving the structure of an application. Data modules in Delphi even exist in versions specific for multitier applications (remote data modules) and server-side HTTP applications (Web data modules).

Finally, remember that you can use the Diagram page of the editor, introduced in Chapter 1, to see a graphical representation of the connections among the components of a data module, as you can see in this example for the MastDet application:



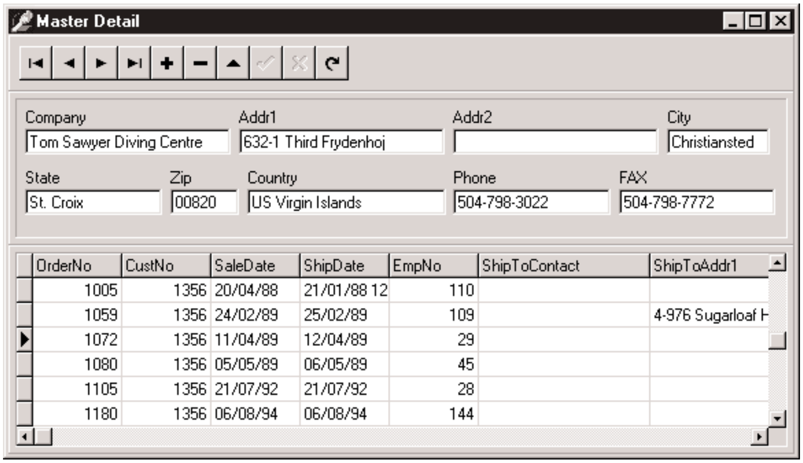
The following is the complete listing (only without the irrelevant positional properties) of the Data Module used by the MastDet program on the CD:

```
object DataModule1: TDataModule1
  OnCreate = DataModule1Create
  object TableCust: TTable
    DatabaseName = 'DBDEMOS'
    TableName = 'customer.db'
  end
  object TableOrd: TTable
    DatabaseName = 'DBDEMOS'
    IndexName = 'CustNo'
    MasterFields = 'CustNo'
    MasterSource = dsCust
    TableName = 'orders.db'
  end
  object dsCust: TDataSource
    DataSet = TableCust
  end
  object dsOrd: TDataSource
    DataSet = TableOrd
  end
end
```

TIP Starting with Delphi 5, you can also create a master/detail structure using the Data Diagram view of a data module.

In Figure 13.3 you can see an example of the main form of the MastDet program at run time. I've placed data-aware controls related to the master table in the upper portion, and I've placed a grid connected with the detail table in the lower portion of the form. This way, for every master record, you immediately see the list of the connected detail record, in this case all the orders by the current client. Each time you select a new customer, the grid below displays only the orders pertaining to that customer.

FIGURE 13.3: The MastDet example at run time



Company	Addr1	Addr2	City
Tom Sawyer Diving Centre	632-1 Third Frydenhoj		Christiansted

State	Zip	Country	Phone	FAX
St. Croix	00820	US Virgin Islands	504-798-3022	504-798-7772

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact	ShipToAddr1
1005	1356	20/04/88	21/01/88 12	110		
1059	1356	24/02/89	25/02/89	109		4-976 Sugarloaf H
1072	1356	11/04/89	12/04/89	29		
1080	1356	05/05/89	06/05/89	45		
1105	1356	21/07/92	21/07/92	28		
1180	1356	06/08/94	06/08/94	144		

A Master/Detail Structure with Queries

The previous example used two tables to build a master/detail form. As an alternative, you can define this type of join using a SQL statement. After setting the master DataSource for the detailed query, you simply set up its SQL statement with a parameter having the same name of the field of the master dataset this data source refers to.

For this example (called Orders), I've joined the ORDERS.DB table with ITEMS.DB, which describes the items of each order. The two tables can be joined using the `OrderNo` field. When you generate the code, the program behaves exactly like the previous one, MastDet. This time, however, the trick is in the SQL statements of the second query object:

```
select OrderNo, ItemNo, PartNo, Qty
from items
where OrderNo = :OrderNo
```

As you can see, this SQL statement uses a parameter, `OrderNo`. This parameter is connected directly to the first query, because the `DataSource` property of `QueryItems` is set to `dsOrders`, which is connected to `QueryOrders`. In other words, the second query is considered to be a data control connected to the first data source. Each time the current record in the first data source changes, the `QueryItems` component is updated, just like any other component connected to `dsOrders`. The field used for the connection, in this case, is the field having the same name as the query parameter.

Other BDE Related Components

Along with Table, Query, StoredProc, and DataSource, other components are on the Data Access page of the Component palette, the BDE page. I'll cover these components in the next chapter, but here is a short summary:

- The Database component is used for transaction control, security, and connection control. It is generally used only to connect to remote databases in client/server applications or to avoid the overhead of connecting to the same database in several forms. The Database component is also used to set a local alias, one used only inside a program. Once this local alias is set to a given path, the Table and Query components of the application can refer to the local database alias. This is much better than replicating the hard-coded path in each DataSet component of the program.

TIP The Borland Database Engine (BDE) uses an alias to refer to a database file or directory. You can define new aliases for databases by using the Database Explorer or the Database Engine Configuration utility. It is also possible to define them by writing code in Delphi that calls the `AddStandardAlias` and `AddAlias` methods of the `Session` global object, followed by a call to `SaveConfigFile` to make the alias persistent. The alternative is the low-level `DbiAddAlias` BDE function. In some of the program of this chapter I'll use the DBDEMOS database alias, which refers to Delphi's demo database, installed by default in the `C:\Program Files\Common Files\Borland Shared\Data` directory.

- The Session component provides global control over database connections for an application, including a list of existing databases and aliases and an event to customize database login.
- The BatchMove component is used to perform batch operations, such as copying, appending, updating, or deleting values, on one or more databases.
- The UpdateSQL component allows you to write SQL statements to perform various update operations on the dataset, when using a read-only query (that is, when working with a complex query). This component is used as the value of the `UpdateObject` property of tables or queries.

Using Data-Aware Controls

Once you've set up the proper data-access components, you can build a user interface to let a user view the data and eventually edit it. Delphi provides many components that resemble the usual Windows controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all of these components in the Data Controls page of the Delphi Component palette.

All of these components are connected to a data source using the corresponding property, `DataSource`. Some of them relate to the entire dataset, such as the DBGrid and DBNavigator components, while the others refer to a specific field of the data source, as indicated by the `DataField` property. Once you select the `DataSource` property, the `DataField` property will have a list of values available in the drop-down combo box of the Object Inspector.

NOTE In Chapter 18, "Writing Database Components," we'll discuss the technical details of these controls, as we'll see how to write custom data-aware components.

Notice that all the data-aware components are totally unrelated to the data-access technology, provided the data-access component inherits from `TDataSet`. This means that your investment on the user interface is totally preserved when you change the data-access technology. What is true, however, is that some of the lookup components and an extended use of the DBGrid, displaying a lot of data, only make more sense when working with local data, and should generally be avoided in a client/server situation, as we'll see in the next chapter.

Data in a Grid

The DBGrid is a grid capable of displaying a whole table at once. It allows scrolling and navigation, and you can edit the grid's contents. It is an extension of the other Delphi grid controls.

You can customize the DBGrid by setting the various flags of its Options property and modifying its Columns collection. The grid allows a user to navigate the data, using the scrollbars, and perform all the mayor actions. A user can edit the data directly, insert a new record in a given position by pressing the Insert key, append a new record at the end by going to the last record and pressing the Down arrow key, and delete the current record by pressing Ctrl+Del.

The Columns property is a collection where you can choose the fields of the table you want to see in the grid and set column and title properties (color, font, width, alignment, caption, and so on) for each field. Some of the more advanced properties, such as ButtonStyle and DropDownRows, can be used to provide custom editors for the cells of a grid or a drop-down list of values (indicated in the PickList property of the column).

An alternative to the DBGrid is the DBCtrlGrid component, a multirecord grid that can host panels with other data-aware controls. These controls are duplicated in each panel for each record of the dataset. I'll discuss the DBCtrlGrid control at the end of this chapter.

DBNavigator and Dataset Actions

DBNavigator is a collection of buttons used to navigate and perform actions on the database. You can disable some of the buttons of the DBNavigator control, by removing some of the elements of the VisibleButtons set.

The buttons perform basic actions on the connected dataset, so you can easily replace them with your own toolbar, particularly if you use an ActionList component with the predefined database actions provided by Delphi. In this case, in fact, you get all the standard behaviors, but you'll also see the various buttons enabled only when their action is legitimate.

TIP If you use the standard actions, you can avoid connecting them to a specific DataSource component, and the actions will be applied to the dataset connected to the visual control that currently has the input focus. This way a single toolbar can be used for multiple datasets displayed by a form.

Text-Based Data-Aware Controls

There are multiple text-oriented components:

- DBText displays the contents of a field that cannot be modified by the user. It is a data-aware Label graphical control. It can be very useful, but users might confuse this control with the plain labels that indicate the content of each field-based control.

- DBEdit lets the user edit a field (change the current value) using an Edit control. At times, you might want to disable editing and use a DBEdit as if it were a DBText, but highlighting the fact that this is data coming from the database.
- DBMemo lets the user see and modify a large text field, eventually stored in a memo or BLOB (binary large object) field. It resembles the Memo component and has full editing capabilities, but all the text is rendered in a single font.
- DBRichEdit is a component that lets the user edit a formatted text file; it is based on a RichEdit Windows common control and, in contrast to DBMemo, it allows text with multiple fonts and paragraph styles.

List-Based Data-Aware Controls

For letting a user choose a value in a predefined list (which reduces input errors), you can use many different components. DBListBox, DBComboBox, and DBRadioGroup are similar, providing a list of strings in the `Items` property, but they do have some differences:

- The DBListBox component allows selection of predefined items (“closed selection”), but not text input, and can be used to list many elements. Generally it’s best to show only about six or seven items, to avoid using up too much space on the screen.
- The DBComboBox component can be used both for closed selection and for user input. The `csDropDown` style of the DBComboBox, in fact, allows a user to enter a new value, besides selecting one of the available ones. The component also uses a smaller area of the form because the drop-down list is usually displayed only on request.
- The DBRadioGroup component presents radio buttons (which permit only one selection), allows only closed selection, and should be used only for a limited number of alternatives. A nice features of this component is that the values displayed can be exactly those you want to insert in the database, but you can also choose to provide some sort of mapping. The values of the user interface (some descriptive strings stored in the `Items` property) will map to corresponding values stored in the database (some numeric or character-based codes listed in the `Values` property). For example, you can map some numeric codes indicating departments to a few descriptive strings:

```
object DBRadioGroup1: TDBRadioGroup
  Caption = 'Department'
  DataField = 'Department'
  DataSource = DataSource1
  Items.Strings = (
    'Sales'
    'Accounting'
    'Production'
    'Management')
    Values.Strings = (
      '1'
      '2'
      '3'
      '4')
  end
```

A slightly different component is the DBCheckBox, used to show and toggle an option, corresponding to a Boolean field. It is a limited list, because it has only two possible values, plus the undetermined state for fields with null values. You can determine which are the values to send back to the database by setting the `ValueChecked` and `ValueUnchecked` properties of this component.

The usage of a DBRadioGroup control, with the settings discussed above, and a DBCheckBox control is highlighted by the DbAware example.

Creating Local Tables with FieldDefs

The DbAware example would be a rather simple program if it didn't have an extra feature: It can create a new table for the DBDEMOS database. Delphi allows you to set the definition of the fields of a table—its internal structure—at design time, using the collection editor of the `FieldDefs` property. Once you've defined the fields, you can then right-click the table component at design time and select the Create Table command.

This list of field definitions is generally extracted from the database, but if you set the `StoreDefs` property of the table to `True`, it will be saved in the DFM file along with the other table properties. The effect of the `StoreDefs` property is more complex than it seems at first. If you right-click the form, you'll notice that its local menu offers an Update Table Definition option, along with the expected Delete Table and Rename Table. That is, you can store the field definitions locally, but if the structure of the physical table changes, you should update this definition as well. Until Delphi 4, the field definitions were invariably loaded from the database table at run time; now you can preload them, speeding up the table opening. However, if the local and the actual table definitions do not match, you can get in trouble.

In the DbAware example, I've used this technique to create a new database table, called Workers, which stores data about the employees of a company. This is the definition of the fields of the table, along with the other key properties:

```
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  FieldDefs = <
    item
      Name = 'LastName'
      DataType = ftString
```

Continued on next page

```
          Size = 20
        end
        item
          Name = 'FirstName'
          DataType = ftString
          Size = 20
        end
        item
          Name = 'Department'
          DataType = ftSmallint
        end
        item
          Name = 'Branch'
          DataType = ftString
          Size = 20
        end
        item
          Name = 'Senior'
          DataType = ftBoolean
        end
        item
          Name = 'HireDate'
          DataType = ftDate
        end>
      StoreDefs = True
      TableName = 'Workers'
    end
```

Regardless of the fact you might have created the table at design time, the program must do so the first time it is executed on a different computer. In practice, when the program starts, it checks whether the table already exists and creates one if it doesn't:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not Table1.Exists then
    Table1.CreateTable;
  Table1.Open;
end;
```

Finally, the program has some code to fill in the table with random values. As this is kind of tedious but not too complex, I won't discuss the details here, but let you look at the source code of the DbAware example if you are interested. I'll use the table produced by this program in other examples in this book, so you might want to run it once and create the table anyway.

Using Lookup Controls

If the list of values is extracted from another dataset, then instead of the DBListBox and DBComboBox controls you should use the specific DBLookupListBox or DBLookupComboBox components. These components are used every time you want to select for a field a record of another dataset.

For example, if you build a standard form for taking orders, the orders dataset will generally have a field hosting a number indicating the customer who made the order. Working directly with the customer number is not the most natural way; most users will prefer to work with customer names. However, in the database, the names of the customers are stored in a different table, to avoid duplicating the customer data for each order by the same customer. To get around such a situation, with local databases or small lookup tables, you can use a DBLookupComboBox control. (This technique doesn't port very well to client/server architecture with large lookup tables, as discussed in the next chapter.)

The DBLookupComboBox component can be connected to two data sources at the same time, one source containing the actual data and a second containing the display data. Basically, I've built a standard form using the ORDERS.DB tables of the DBDEMOS database, with several DBEdit controls (you can as well use the Database Form Wizard to build this plain form). The example actually uses a Query component selecting most fields of the orders table.

At this point we want to remove the standard DBEdit component connected to the customer number and replace it with a DBLookupComboBox component (and a DBText component for understanding what exactly is going on). The lookup component (and the DBText) is connected with the DataSource for the order and with the CustNo field. To let the lookup component show the information extracted from another table, CUSTOMER.DB, we need to add another table component referring to it, and new data source connected to the table.

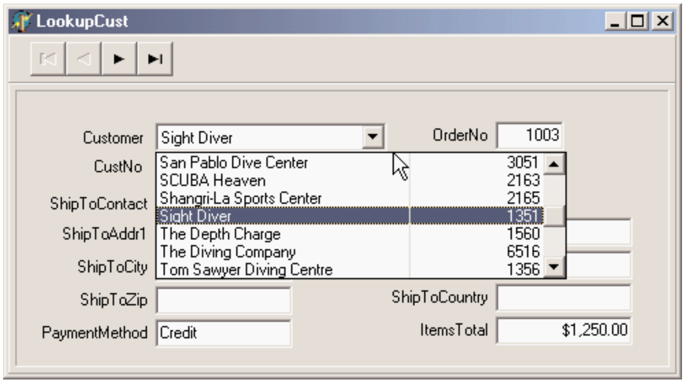
For the program to work, you need to set several properties of the `DBLookupComboBox1` component. Here is a list of the relevant values:

```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSourceOrders
  KeyField = 'CustNo'
  ListField = 'Company;CustNo'
  ListSource = DataSourceCustomer
  DropDownWidth = 300
end
```

The first two properties determine the main connection, as usual. The other three properties determine the secondary source (`ListSource`), the field used for the join (`KeyField`), and the information to display (`ListField`). Besides entering the name of a single field, you can provide multiple fields, as I've done in the example. Only the first field is displayed as combo box text, but if you set a large value for the `DropDownWidth` property, the pull-down list of the combo box will include multiple columns of data. You can see this output in Figure 13.4.

FIGURE 13.4:

The output of the CustLookup example, with the DBLookupComboBox showing multiple fields in its drop-down list



Customer: Sight Diver
OrderNo: 1003
CustNo:
ShipToContact:
ShipToAddr1:
ShipToCity:
ShipToZip:
ShipToCountry:
PaymentMethod: Credit
ItemsTotal: $1,250.00

San Pablo Dive Center	3051
SCUBA Heaven	2163
Shangri-La Sports Center	2165
Sight Diver	1351
The Depth Charge	1560
The Diving Company	6516
Tom Sawyer Diving Centre	1356

TIP If you set the index of the table connected with the DBLookupComboBox to the `Company` field, the drop-down list will show the companies in alphabetical order instead of customer-number order. This is what I've done in the example.

What about the code of this program? Well, there is none. Everything works just by setting the correct properties. The three joined data sources do not need custom code. This demonstrates that using master/detail and lookup connections can be very fast to set up and very efficient. The only real drawback is that these techniques, particularly the lookup, cannot be used when the number of records becomes too large, particularly in a networked or client/server environment. Moving hundreds of thousands of records just to make a nice-looking lookup combo box probably won't be very effective.

NOTE In Delphi 6, both the `TDBLookupComboBox` and `TDBLookupListBox` controls have a `NullValueKey` property, which indicates the shortcut that can be used to set the value to null, by calling the `Clear` method of the corresponding field.

Graphical Data-Aware Controls

Finally, Delphi includes two graphical data-aware controls:

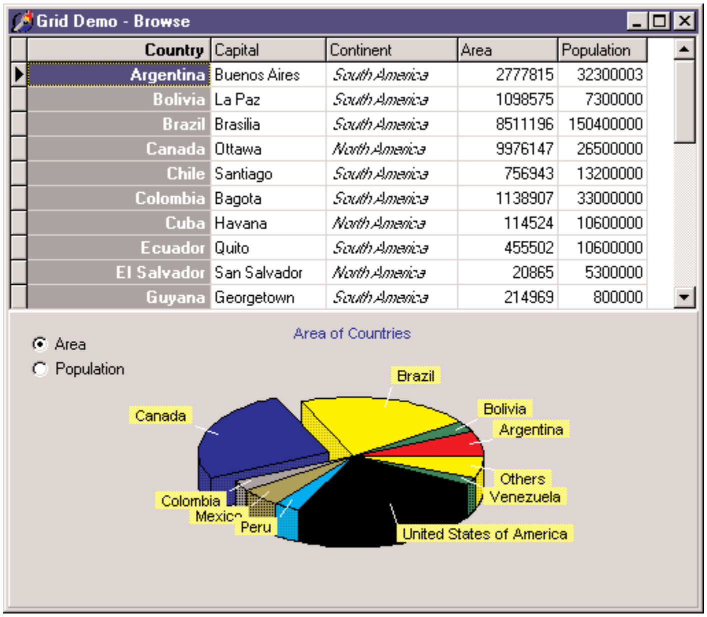
- DBImage is an extension of an Image component that shows a picture stored in a BLOB field (provided the database uses a graphic format that the Image component supports, such as BMP and JPEG).

- DBChart is a data-aware business graphic component or the data-aware version of the TeeChart control built by David Berneda.

To demonstrate the use of the DBChart control, I've added this component to a simple example showing a data grid. The application, called ChartDB, shows a pie chart with the surface of each country of the COUNTRY.DB table, as you can see in Figure 13.5.

FIGURE 13.5:

The output of the ChartDB example, which is based on the `TDbChart` control



Grid Demo - Browse

Country	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300003
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000

● Area
○ Population

Area of Countries

The program has almost no code, as all the settings can be done using the specific component editor, which has several options but is quite easy to use. Here are some of the key properties of the component, taken from the form description:

```
object DBChart1: TDBChart
  Legend.Visible = False
  Align = alClient
  object Series1: TPieSeries
    Marks.ArrowLength = 8
    Marks.Visible = True
    DataSource = Table1
    XLabelsSource = 'Name'
    ExplodeBiggest = 3
    OtherSlice.Style = poBelowPercent
    OtherSlice.Text = 'Others'
    OtherSlice.Value = 2
    PieValues.ValueSource = 'Area'
  end
end
```

What I've done is show the area field as the data source for the pie chart (the PieValues .ValueSource property of the series), use the name field for the labels (the XLabelsSource property of the series), and condense all the countries with a value below 2 percent in a single section indicated as Others (the OtherSlide subproperties).

As a minor addition to the code, I've added two radio buttons you can use to toggle between the area and the population. The code of the two radio buttons simply sets the source of the series, after casting it to the proper series type, as in:

```
procedure TForm1.RadioPopulationClick(Sender: TObject);
begin
  DBChart1.Title.Text [0] := 'Population of Countries';
  (DBChart1.Series [0] as TPieSeries).PieValues.ValueSource := 'Population';
end;
```

The DataSet Component

Instead of focusing right away on the use of a specific dataset, I prefer starting with a generic introduction of the features of the TDataSet class, which are shared by all inherited data-access classes. The DataSet component is a very complex one, so I won't list all of its capabilities but only discuss its core elements.

The idea behind this component is to provide access to a series of records that are read from some source of data, kept in internal buffers (for performance reasons), and eventually modified by a user, with the possibility of writing back changes to the persistent storage. This approach is generic enough to be applied to different types of data (even non-database data) but has a few rules. First, there can be only one active record at a time, so if you need to access data in multiple records, you must move to each of them, read the data, then move again, and so on. You'll find an example of this and related techniques in the section about navigation.

Second, you can edit only the active record: you cannot modify a set of records at the same time, as you can in a relational database. Moreover, you can modify data in the active buffer only after you explicitly declare you want to do so, by giving the Edit command to the dataset. You can also use the Insert command to create a new blank record, and close both operations (insert or edit) by giving a Post command.

Other interesting elements of a dataset I will explore in the following sections are its status (and the status change events), navigation and record positions, and the role of the field objects. As a summary of the capabilities of the DataSet component, I've included the public methods of its class in Listing 13.1 (the code has been edited and commented for clarity). Not all of these methods are directly used everyday, but I decided to keep them all in the listing. In

Chapter 18, I'll also discuss the virtual methods of the protected portion of the class, which we'll need to override to build custom dataset components.

Listing 13.1: The public interface of the *TDataSet* class (excerpted)

```
TDataSet = class(TComponent, IProviderSupport)
...
public
  // create and destroy, open and close
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure Open;
  procedure Close;
  property BeforeOpen: TDataSetNotifyEvent read FBeforeOpen write FBeforeOpen;
  property AfterOpen: TDataSetNotifyEvent read FAfterOpen write FAfterOpen;
  property BeforeClose: TDataSetNotifyEvent
    read FBeforeClose write FBeforeClose;
  property AfterClose: TDataSetNotifyEvent read FAfterClose write FAfterClose;

  // status information
  function IsEmpty: Boolean;
  property Active: Boolean read GetActive write SetActive default False;
  property State: TDataSetState read FState;
  function ActiveBuffer: PChar;
  property IsUniDirectional: Boolean
    read FIsUniDirectional write FIsUniDirectional default False;
  function UpdateStatus: TUpdateStatus; virtual;
  property RecordSize: Word read GetRecordSize;
  property ObjectView: Boolean read FObjectView write SetObjectView;
  property RecordCount: Integer read GetRecordCount;
  function IsSequenced: Boolean; virtual;
  function IsLinkedTo(DataSource: TDataSource): Boolean;

  // datasource
  property DataSource: TDataSource read GetDataSource;
  procedure DisableControls;
  procedure EnableControls;
  function ControlsDisabled: Boolean;

  // fields, including blobs, details, calculated, and more
  function FieldByName(const FieldName: string): TField;
  function FindField(const FieldName: string): TField;
  procedure GetFieldList(List: TList; const FieldNames: string);
  procedure GetFieldNames(List: TStrings);
  property FieldCount: Integer read GetFieldCount;
  property FieldDefs: TFieldDefs read FFieldDefs write SetFieldDefs;
  property FieldDefList: TFieldDefList read FFieldDefList;
  property Fields: TFields read FFields;
  property FieldList: TFieldList read FFieldList;
  property FieldValues[const FieldName: string]: Variant
  read GetFieldValue write SetFieldValue; default;
property AggFields: TFields read FAggFields;
property DataSetField: TDataSetField
  read FDataSetField write SetDataSetField;
property DefaultFields: Boolean read FDefaultFields;
procedure ClearFields;
function GetBlobFieldData(FieldNo: Integer;
  var Buffer: TBlobByteData): Integer; virtual;
function CreateBlobStream(Field: TField;
  Mode: TBlobStreamMode): TStream; virtual;
function GetFieldData(Field: TField;
  Buffer: Pointer): Boolean; overload; virtual;
procedure GetDetailDataSets(List: TList); virtual;
procedure GetDetailLinkFields(MasterFields, DetailFields: TList); virtual;
function GetFieldData(FieldNo: Integer;
  Buffer: Pointer): Boolean; overload; virtual;
function GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean):
  Boolean; overload; virtual;
property AutoCalcFields: Boolean
  read FAutoCalcFields write FAutoCalcFields default True;
property OnCalcFields: TDataSetNotifyEvent
  read FOnCalcFields write FOnCalcFields;

// position, movement
procedure CheckBrowseMode;
procedure First;
procedure Last;
procedure Next;
procedure Prior;
function MoveBy(Distance: Integer): Integer;
property RecNo: Integer read GetRecNo write SetRecNo;
property Bof: Boolean read FBOF;
property Eof: Boolean read FEOF;
procedure CursorPosChanged;
property BeforeScroll: TDataSetNotifyEvent
  read FBeforeScroll write FBeforeScroll;
property AfterScroll: TDataSetNotifyEvent
  read FAfterScroll write FAfterScroll;

// bookmarks
procedure FreeBookmark(Bookmark: TBookmark); virtual;
function GetBookmark: TBookmark; virtual;
function BookmarkValid(Bookmark: TBookmark): Boolean; virtual;
procedure GotoBookmark(Bookmark: TBookmark);
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; virtual;
property Bookmark: TBookmarkStr read GetBookmarkStr write SetBookmarkStr;

// find, locate
function FindFirst: Boolean;
function FindLast: Boolean;
function FindNext: Boolean;
function FindPrior: Boolean;
property Found: Boolean read GetFound;
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean; virtual;
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant; virtual;

// filtering
property Filter: string read FFilterText write SetFilterText;
property Filtered: Boolean read FFiltered write SetFiltered default False;
property FilterOptions: TFilterOptions
  read FFilterOptions write SetFilterOptions default [];
property OnFilterRecord: TFilterRecordEvent
  read FOnFilterRecord write SetOnFilterRecord;

// refreshing, updating
procedure Refresh;
property BeforeRefresh: TDataSetNotifyEvent
  read FBeforeRefresh write FBeforeRefresh;
property AfterRefresh: TDataSetNotifyEvent
  read FAfterRefresh write FAfterRefresh;
procedure UpdateCursorPos;
procedure UpdateRecord;
function GetCurrentRecord(Buffer: PChar): Boolean; virtual;
procedure Resync(Mode: TResyncMode); virtual;

// editing, inserting, posting, and deleting
property CanModify: Boolean read GetCanModify;
property Modified: Boolean read FModified;
procedure Append;
procedure Edit;
procedure Insert;
procedure Cancel; virtual;
procedure Delete;
procedure Post; virtual;
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
procedure SetFields(const Values: array of const);

// events related to editing, inserting, posting, and deleting
property BeforeInsert: TDataSetNotifyEvent
  read FBeforeInsert write FBeforeInsert;
property AfterInsert: TDataSetNotifyEvent
  read FAfterInsert write FAfterInsert;
property BeforeEdit: TDataSetNotifyEvent read FBeforeEdit write FBeforeEdit;
property AfterEdit: TDataSetNotifyEvent read FAfterEdit write FAfterEdit;
property BeforePost: TDataSetNotifyEvent read FBeforePost write FBeforePost;
property AfterPost: TDataSetNotifyEvent read FAfterPost write FAfterPost;
property BeforeCancel: TDataSetNotifyEvent
  read FBeforeCancel write FBeforeCancel;
property AfterCancel: TDataSetNotifyEvent
```

```
      read FAfterCancel write FAfterCancel;
    property BeforeDelete: TDataSetNotifyEvent
      read FBeforeDelete write FBeforeDelete;
    property AfterDelete: TDataSetNotifyEvent
      read FAfterDelete write FAfterDelete;
    property OnDeleteError: TDataSetErrorEvent
      read FOnDeleteError write FOnDeleteError;
    property OnEditError: TDataSetErrorEvent
      read FOnEditError write FOnEditError;
    property OnNewRecord: TDataSetNotifyEvent
      read FOnNewRecord write FOnNewRecord;
    property OnPostError: TDataSetErrorEvent
      read FOnPostError write FOnPostError;

    // support, utilities
    function Translate(Src, Dest: PChar;
      ToOem: Boolean): Integer; virtual;
    property Designer: TDataSetDesigner read FDesigner;
    property BlockReadSize: Integer read FBlockReadSize write SetBlockReadSize;
    property SparseArrays: Boolean read FSparseArrays write SetSparseArrays;
  end;
```

The Status of a Dataset

When you operate on a dataset in Delphi, you can work in different states, indicated by a specific State property, which can assume several different values:

dsBrowse indicates that the dataset is in normal browse mode, used to look at the data and scan the records.

dsEdit indicates that the dataset is in edit mode. A dataset enters this state when the program calls the Edit method or the DataSource has the AutoEdit property set to True, and the user starts editing a data-aware control, such as a DBGrid or DBEdit. When the changed record is posted, the dataset exits the dsEdit state.

dsInsert indicates that a new record is being added to the dataset. Again, this might happen when calling the Insert method, moving to the last line of a DBGrid, or using the corresponding command of the DBNavigator component.

dsInactive is the state of a closed dataset.

dsSetKey indicates that we are preparing a search on the dataset. This is the state between a call to the SetKey method and a call to the GotoKey or GotoNearest methods (see the Search example later in this chapter).

dsCalcFields is the state of a dataset while a field calculation is taking place; that is, during a call to an OnCalcFields event handler. Again, I'll show this in an example.

dsNewValue, dsOldValue, and dsCurValue are the states of a dataset when an update of the cache is in progress.

dsFilter is the state of a dataset while setting a filter; that is, during a call to an OnFilterRecord event handler.

In simple examples, the transitions between these states are handled automatically, but it is important to understand them because there are many events referring to the state transitions. For example, every dataset fires events before and after any state change. When a program requests an Edit operation, the component fires the BeforeEdit event just before entering in edit mode (an operation you can stop by raising an exception). Immediately after entering edit mode, the dataset receives the AfterEdit event. After the user has finished editing and requests to store the data, executing the Post command, the dataset fires a BeforePost event, which can be used to check the input before sending the data to the database, and an AfterPost event after the operation has been successfully completed.

Another more general state-change tracking technique is to handle the OnStateChange event of the DataSource component. As a very simple example you can show the current status with code like this:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  strStatus: string;
begin
  case Table1.State of
    dsBrowse: strStatus := 'Browse';
    dsEdit: strStatus := 'Edit';
    dsInsert: strStatus := 'Insert';
  else
    strStatus := 'Other state';
  end;
  StatusBar.Panels[0].Text := strStatus;
end;
```

The code considers only the three most common states a dataset component, ignoring the inactive state and other special cases.

The Fields of a Dataset

I mentioned earlier that a dataset has only one record that is the current, or active, one. The record is stored in a buffer, and you can operate on it with some generic methods, but to access the data of the record you need to use the field objects of the dataset. This explains why field components (technically instances of class derived from the TField class) play a fundamental role in every Delphi database application. Data-aware controls are directly connected to these field objects, which correspond to database fields.

By default, Delphi automatically creates the TField components at run time, each time the program opens a dataset component. This is done after reading the metadata associated with the table or the query the dataset refers to. These field components are stored in the Fields array property of a dataset. You can access these values by number (accessing the array directly) or by name (using the FieldByName method). Each field can be used for reading or modifying the data of the current record, using its Value property or type-specific properties, like AsDate, AsString, AsInteger, and so on:

```
var
  strName: string;
begin
  strName := Table1.Fields[0].AsString
  strName := Table1.FieldByName('LastName').AsString
```

Value is a variant type property, so using the type-specific access properties is a little more efficient. The dataset component has also a shortcut property for accessing the variant-type value of a field, the default FieldValues property. Being a default property means you can omit it from the code by applying the square brackets directly to the dataset:

```
strName := Table1.FieldValues ['LastName'];
strName := Table1 ['LastName'];
```

Creating the field components each time a dataset is opened is only a default behavior. As an alternative, you can create the field components at design time, using the Fields editor (double-click a dataset to see the Fields editor in action, or activate its local menu and choose the Fields Editor command). After creating a field for the LastName column of a table, for example, you can refer to its value by applying one of the AsXxx methods to the proper field object:

```
strName := Table1LastName.AsString;
```

Besides being used to access the value of a field, each field object also has properties for controlling visualization and editing of its value, including range of values, edit masks, display format, constraints, and many others. These properties, of course, depend on the type of the field—that it is, on the specific class of the field object. If you create persistent fields you can set some properties at design time, instead of writing code at run time, maybe in the AfterOpen event of the dataset.

NOTE Although the Fields editor is similar to the editors of the collections used by Delphi, fields are not part of a collection. They are components created at design time, listed in its published section of the form class, and available in the drop-down combo box at the top of the Object Inspector.

As you open the Fields editor for a dataset, it appears empty. You have to activate the local menu of this editor to access its capabilities. The simplest operation you can do is to select the Add command, which allows you to add any other fields in the dataset to the list of fields. Figure 13.6 shows the Add Fields dialog box, which lists all the fields that are available in a table. These are the database table fields that are not already present in the list of fields in the editor.

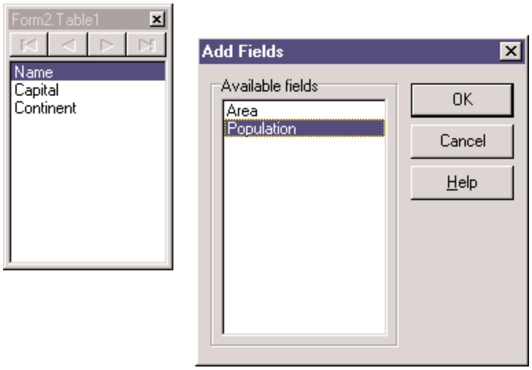


FIGURE 13.6:
The Fields editor with the Add Fields dialog box

The Define command of the Fields editor, instead, lets you define a new calculated field, lookup field, or field with a modified type. In this dialog box, you can enter a descriptive field name, which might include blank spaces. Delphi generates an internal name—the name of the field component—which you can further customize. Next, select a data type for the field. If this is a calculated field or a lookup field, and not just a copy of a field redefined to use a new data type, simply check the proper radio button. We'll see how to define a calculated field in the section "Adding a Calculated Field" and a lookup field in two following sections.

NOTE A `TField` component has both a `Name` property and a `FieldName` property. The `Name` property is the usual component name. The `FieldName` property is either the name of the column in the database table or the name you define for the calculated field. It can be more descriptive than the `Name`, and it allows blank spaces. The `FieldName` property of the `TField` component is copied to the `DisplayLabel` property by default, but this field name can be changed to any suitable text. It is used, among other things, to search a field in the `FieldByName` method of the `TDataSet` class and when using the array notation.

All of the fields that you add or define are included in the Fields editor and can be used by data-aware controls or displayed in a database grid. If a field of the original database table is not in this list, it won't be accessible. When you use the Fields editor, Delphi adds the declaration of the available fields to the class of the form, as new components (much as the Menu Designer adds `TMenuItem` components to the form). The components of the `TField` class, or more specifically its subclasses, are fields of the form, and you can refer to these components directly in the code of your program to change their properties at run time or to get or set their value.

In the Fields editor, you can also drag the fields to change their order. Proper field ordering is particularly important when you define a grid, which arranges its columns using this order.

TIP An even better feature of the Fields editor is that you can drag fields from this editor to the surface of a form and have Delphi automatically create a corresponding data-aware control (such as a DBEdit, a DBMemo, or a DBImage). The type of control created depends on the data type of the field. This is a very fast way to generate custom forms, and I suggest you try it out if you've never used it before. This is my preferred way to build database-related forms, much better than using the Database Form Wizard.

Using Field Objects

Before we look at an example, let's go over the use of the TField class. The importance of this component should not be underestimated. Although it is often used behind the scenes, its role in database applications is fundamental. As I already mentioned, even if you do not define specific objects of this kind, you can always access the fields of a table or a query using their Fields array property, the FieldValues indexed property, or the FieldByName method. Both the Fields property and the FieldByName function return an object of type TField, so you sometimes have to use the as operator to downcast their result to its actual type (like TFloatField or TDateField) before accessing specific properties of these subclasses.

The FieldAcc example is a simple extension of a form generated by the Database Form Wizard. I've added to it three speed buttons in the toolbar panel, accessing various field properties at run time. The first button changes the formatting of the population column of the grid. To do this, we have to access the DisplayFormat property, a specific property of the TFloatField class. For this reason we have to write:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
  (Table1.FieldByName ('Population') as
    TFloatField).DisplayFormat := '###,###,###';
end;
```

When you set field properties related to data input or output, the change applies to every record in the table. When you set properties related to the value of the field, instead, you always refer to the current record only. For example, we can output the population of the current country in a message box by writing:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);
begin
  ShowMessage (string (Table1 ['Name']) +': '+ string (Table1 ['Population']));
end;
```

When you access the value of a field, you can use a series of *As* properties to handle the current field value using a specific data type (if this is available; otherwise, an exception is raised):

```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt;
AsString: string;
AsVariant: Variant;
```

These properties can be used to read or change the value of the field. Changing the value of a field is possible only if the dataset is in edit mode. As an alternative to the *As* properties indicated above, you can access the value of a field by using its `Value` property, which is defined as a `variant`.

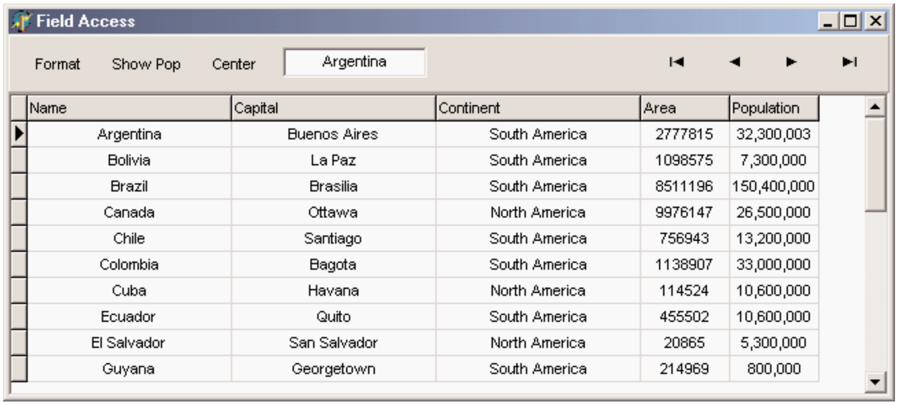
Most of the other properties of the `TField` component, such as `Alignment`, `DisplayLabel`, `DisplayWidth`, and `Visible`, reflect elements of the field's user interface and are used by the various data-aware controls, particularly DBGrid. In the FieldAcc example, clicking the third speed button changes the `Alignment` of every field:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 t7 Table1.FieldCount - 1 do
    Table1.Fields[I].Alignment := taCenter;
end;
```

This affects the output of the DBGrid, and of the DBEdit control I've added to the toolbar, which shows the name of the country. You can see this effect, along with the new display format, in Figure 13.7.

FIGURE 13.7:

The output of the FieldAcc example after the Center and Format buttons have been pressed



Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32,300,003
Bolivia	La Paz	South America	1098575	7,300,000
Brazil	Brasilia	South America	8511196	150,400,000
Canada	Ottawa	North America	9976147	26,500,000
Chile	Santiago	South America	756943	13,200,000
Colombia	Bagota	South America	1138907	33,000,000
Cuba	Havana	North America	114524	10,600,000
Ecuador	Quito	South America	455502	10,600,000
El Salvador	San Salvador	North America	20865	5,300,000
Guyana	Georgetown	South America	214969	800,000

A Hierarchy of Field Classes

There are several field class types in VCL. Delphi automatically uses one of them depending on the data definition in the database, when you open a table at run time or when you use the Fields editor at design time. Table 13.1 shows the complete list of subclasses of the TField class.

TABLE 13.1: The Subclasses of TField

Subclass	Base Class	Definition
TADTField	TObjectField	An ADT (Abstract Data Type) field, corresponding to an object field in an object relational database.
TAggregateField	TField	An aggregate field represents a maintained aggregate. It is used in the ClientDataSet component and discussed in Chapter 14, "Client/Server Programming."
TArrayField	TObjectField	An array of objects in an object relational database.
TAutoIncField	TIntegerField	Whole positive number connected with a Paradox auto-increment field of a table, a special field automatically assigned a different value for each record. Note that Paradox AutoInc fields do not always work perfectly, as discussed in the next chapter.
TBCDField	TNumericField	Real numbers, with a fixed number of digits after the decimal point.
TBinaryField	TField	Generally not used directly. This is the base class of the next two classes.
TBlobField	TField	Binary data and no size limit (BLOB stands for binary large object). The theoretical maximum limit is 2 GB.
TBooleanField	TField	Boolean value.
TBytesField	TBinaryField	Arbitrary data with a large (up to 64 KB characters) but fixed size.
TCurrencyField	TFloatField	Currency values, with the same range as the new Real data type.
TDataSetField	TObjectField	An object corresponding to a separate table in an object relational database.
TDateField	TDateTimeField	Date value.
TDateTimeField	TField	Date and time value.
TFloatField	TNumericField	Floating-point numbers (8 byte).
TFMTBCDField	TNumericField	(New field type in Delphi 6) A true binary-coded decimal (BCD), as opposed to the existing TBCDField type, which converted BCD values to the Currency type. This field type is used automatically only by dbExpress datasets.
TGraphicField	TBlobField	Graphic of arbitrary length.
TGuidField	TStringField	A field representing a COM Globally Unique Identifier, part of the ADO support.

Continued on next page

TABLE 13.1 continued: The Subclasses of TField

Subclass	Base Class	Definition
TIDispatchField	TInterfaceField	A field representing pointers to IDispatch COM interfaces, part of the ADO support.
TIntegerField	TNumericField	Whole numbers in the range of long integers (32 bits).
TInterfacedField	TField	Generally not used directly. This is the base class of fields that contain pointers to interfaces (IUnknown) as data.
TLargeIntField	TIntegerField	Very large integers (64 bit).
TMemoField	TBlobField	Text of arbitrary length.
TNumericField	TField	Generally not used directly. This is the base class of all the numeric field classes.
TObjectField	TField	Generally not used directly. The base class for the fields providing support for object relational databases.
TReferenceField	TObjectField	A pointer to an object in an object relational database.
TSmallIntField	TIntegerField	Whole numbers in the range of integers (16 bits).
TSQLTimeStampField	TField	(New field type in Delphi 6) Supports the date/time representation used in dbExpress drivers
TStringField	TField	Text data of a fixed length (up to 8192 bytes).
TTimeField	TDateTimeField	Time value.
TVarBytesField	TBytesField	Arbitrary data, up to 64 KB characters. Very similar to the TBytes-Field base class.
TVariantField	TField	A field representing a variant data type, part of the ADO support.
TWideStringField	TStringField	A field representing a Unicode (16 bits per character) string.
TWordField	TIntegerField	Whole positive numbers in the range of words or unsigned integers (16 bits).

The availability of any particular field type, and the correspondence with the data definition, depends on the database in use. This is particularly true for the new field types that provide support for object relational databases.

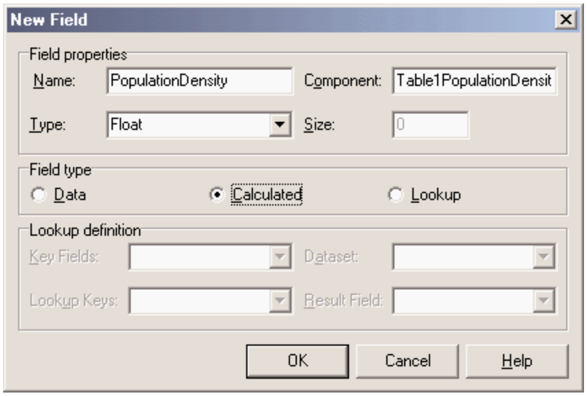
Adding a Calculated Field

Now that you've been introduced to TField objects and seen an example of their run-time use, it is time to build a simple example based on the declaration of field objects at design time using the Fields editor. We can start again from the first example we built, GridDemo, and add a calculated field. The COUNTRY.DB database table we are accessing has both the population and the area of each country, so we can use this data to compute the population density.

To build the new example, named Calc, select the Table component in the form, and open the Fields editor. In this editor, choose the Add Fields command, and select some of the fields. (I've decided to include them all.) Now select the New Field command, and enter a proper name and data type (`TFloatField`) for the new calculated field, as you can see in Figure 13.8.

FIGURE 13.8:

The definition of a calculated field in the Calc example



New Field

Field properties

Name: PopulationDensity

Component: Table1PopulationDensit

Type: Float

Size: 0

Field type

○ Data ● Calculated ○ Lookup

WARNING It is obvious that as you create some field components at design time using the Fields editor, the fields you skip won't get a corresponding object. What might not be obvious is that the fields you skip will not be available even at run time, with Fields or `FieldByName`. When a program opens a table at run time, if there are no design-time field components, Delphi creates field objects corresponding to the table definition. If there are some design-time fields, however, Delphi uses those fields without adding any extra ones.

Of course, we also need to provide a way to calculate the new field. This is accomplished in the `OnCalcFields` event of the Table component, which has the following code (at least in a first version):

```
procedure TForm2.Table1CalcFields(DataSet: TDataSet);
begin
  Table1PopulationDensity.Value := Table1Population.Value / Table1Area.Value;
end;
```

NOTE Calculated fields are computed for each record and recalculated each time the record is loaded in an internal buffer, invoking the `OnCalcFields` event over and over again. For this reason, a handler of this event should be extremely fast to execute, and cannot alter the status of the dataset, by accessing different records. A more time-efficient (but less memory-efficient) version of a calculated field is provided by the ClientDataSet component with "internally calculated" fields, which are evaluated only once, when they are loaded, with the result stored in memory for future requests.

Everything fine? Not at all! If you enter a new record and do not set the value of the population and area, or if you accidentally set the area to zero, the division will raise an exception, making it quite problematic to continue using the program. As an alternative, we could have handled every exception of the division expression and simply set the resulting value to zero:

```
try
  Table1PopulationDensity.Value := Table1Population.Value / Table1Area.Value;
except
  on Exception do
    Table1PopulationDensity.Value := 0;
end;
```

However, we can do even better. We can check if the value of the area is defined—if it is not null—and if it is not zero. It is better to avoid using exceptions when you can anticipate the possible error conditions:

```
if not Table1Area.IsNull and
    (Table1Area.Value <> 0) then
  Table1PopulationDensity.Value := Table1Population.Value / Table1Area.Value
else
    Table1PopulationDensity.Value := 0;
```

The code of the Table1CalcFields method above (in each of the three versions) accesses some fields directly. This is possible because I used the Fields editor, and it automatically created the corresponding field declarations, as you can see in this excerpt of the interface declaration of the form:

```
type
  TCalcForm = class(TForm)
    Table1: TTable;
    Table1PopulationDensity: TFloatField;
    Table1Area: TFloatField;
    Table1Population: TFloatField;
    Table1Name: TStringField;
    Table1Capital: TStringField;
    Table1Continent: TStringField;
    procedure Table1CalcFields(DataSet: TDataset);
    ...
```

Each time you add or remove fields in the Fields editor, you can see the effect of your action immediately in the grid present in the form. Of course, you won't see the values of a calculated field at design time; they are available only at run time, because they result from the execution of compiled Pascal code.

Since we have defined some components for the fields, we can use them to customize some of the visual elements of the grid. For example, to set a display format that adds a comma to separate thousands, we can use the Object Inspector to change the DisplayFormat property of some field components to "###,###,###". This change has an immediate effect on the grid at design time.

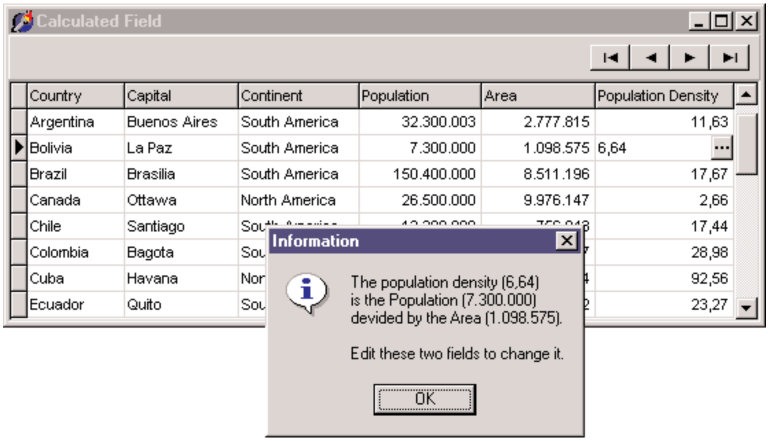
NOTE The display format I've just mentioned (and used in the previous example) uses the Windows International Settings to format the output. When Delphi translates the numeric value of this field to text, the comma in the format string is replaced by the proper `ThousandSeparator` character. For this reason, the output of the program will automatically adapt itself to different International Settings. On computers that have the Italian configuration, for example, the comma is replaced by a period.

After working on the table components and the fields, I've customized the DBGrid using its `Columns` property editor. I've set the Population Density column to read-only and set its `ButtonStyle` property to cbsEllipsis, to provide a custom editor. When you set this value, a small button with an ellipsis is displayed when the user tries to edit the grid cell. Pressing the button invokes the `OnEditButtonClick` event of the DBGrid:

```
procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  MessageDlg (Format (
    'The population density (%.2n)'#13 +
    'is the Population (%.0n)'#13 +
    'divided by the Area (%.0n).'#13#13 +
    'Edit these two fields to change it.',
    [Table1PopulationDensity.AsFloat,
    Table1Population.AsFloat,
    Table1Area.AsFloat]),
    mtInformation, [mbOK], 0);
end;
```

Actually, I haven't provided a real editor, but rather a message describing the situation, as you can see in Figure 13.9, which shows the values of the calculated fields. To create an editor, you might build a secondary form to handle special data entries.

FIGURE 13.9: The output of the Calc example. Notice the Population Density calculated column, the ellipsis button, and the message displayed when you select it.



Country	Capital	Continent	Population	Area	Population Density
Argentina	Buenos Aires	South America	32.300.003	2.777.815	11,63
Bolivia	La Paz	South America	7.300.000	1.098.575	6,64 ...
Brazil	Brasilia	South America	150.400.000	8.511.196	17,67
Canada	Ottawa	North America	26.500.000	9.976.147	2,66
Chile	Santiago	[illegible]	[illegible]	[illegible]	17,44
Colombia	Bagota	Sou		7	28,98
Cuba	Havana	Nor		4	92,56
Ecuador	Quito	Sou		2	23,27

The population density (6,64)
is the Population (7.300.000)
devided by the Area (1.098.575).

Edit these two fields to change it.

Lookup Fields

As an alternative to placing a DBLookupComboBox component in a form (discussed earlier in this chapter under "Using Lookup Controls"), we can also define a lookup field, which can be displayed with a drop-down lookup list inside a DBGrid component. We've seen that to add a fixed selection to a DBGrid, we can simply edit the `PickList` subproperty of the `Columns` property. To customize the grid with a live lookup, instead, we have to define a lookup field using the Fields editor.

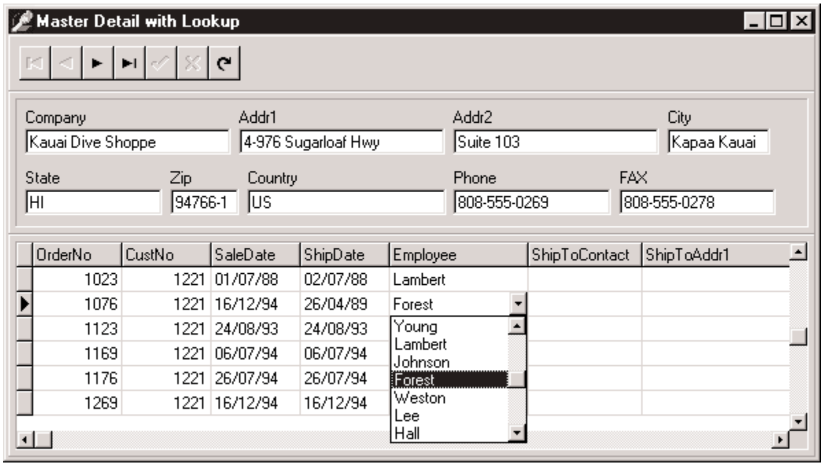
As an example, I've built the FieldLookup program, which has a grid displaying orders with a lookup field to display the name of the employee who took the order, instead of the code number of this employee. To accomplish this, I added to the data module a Table component referring to the EMPLOYEE.DB table. Then I opened the Fields editor for the ORDERS table and added all the fields. I selected the `EmpNo` field and set its `Visible` property to False, to remove it from the grid (we cannot remove it altogether, because it is used to build the cross-reference with the corresponding field of the EMPLOYEE table).

Now it is time to define the lookup field. If you've followed the preceding steps, you can use the Fields editor of the ORDERS table and select the New Field command, obtaining the New Field dialog box. (As an alternative in Delphi 5, it's possible to use the Diagram page of the editor, drop the two tables there, and drag a lookup relation from the ORDERS table to the EMPLOYEE table, connecting the two in the resulting New Field dialog box. In Delphi 6, though, the lookup relation button is still part of the Diagram page but doesn't seem to be working at all.)

The values you specify in the New Lookup Field dialog box will affect the properties of a new `TField` added to the table, as demonstrated by the DFM description of the field:

```
object Table2Employee: TStringField
  FieldKind = fkLookup
  FieldName = 'Employee'
  LookupDataSet = Table2
  LookupKeyFields = 'EmpNo'
  LookupResultField = 'LastName'
  KeyFields = 'EmpNo'
  Size = 30
  Lookup = True
end
```

This is all that is needed to make the drop-down list work (see Figure 13.10) and to view the value of the cross-references field at design time, too. Notice that there is no need to customize the `Columns` property of the grid, because the drop-down button and the value of seven rows are taken by default. This doesn't mean you cannot use this property to further customize these and other visual elements of the grid.



Company	Addr1	Addr2	City
Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai

State	Zip	Country	Phone	FAX
HI	94766-1	US	808-555-0269	808-555-0278

OrderNo	CustNo	SaleDate	ShipDate	Employee	ShipToContact	ShipToAddr1
1023	1221	01/07/88	02/07/88	Lambert		
1076	1221	16/12/94	26/04/89	Forest		
1123	1221	24/08/93	24/08/93			
1169	1221	06/07/94	06/07/94			
1176	1221	26/07/94	26/07/94			
1269	1221	16/12/94	16/12/94			

FIGURE 13.10: The output of the FieldLookup example, with the drop-down list inside the grid displaying values taken from another database table

Handling Null Values with Field Events

Beyond a few interesting properties, the field objects have a few key events. The `OnValidate` event can be used to provide extended validation of the value of a field, and should be used whenever you need a complex rule that the ranges and constraints provided by the field cannot express. This event is triggered before the data is written to the record buffer, whereas the `OnChange` event is fired soon after the data has been written.

Two more events, `OnGetText` and `OnSetText`, can be used to customize the output of a field. These two events are extremely powerful: they allow you to use data-aware controls even when the representation of a field you want to display is different from the one Delphi will provide by default.

An example of the use of these events is the handling of null value. On SQL servers, storing an empty value or a null value for a field are two separate operations. The latter tends to be more correct, but Delphi by default uses empty values and displays the same output for an empty or a null field. Although this can be useful in general for strings and numbers, it becomes extremely important for dates, where it is hard to set a reasonable default value and where if the user blanks out the field you might have an invalid input.

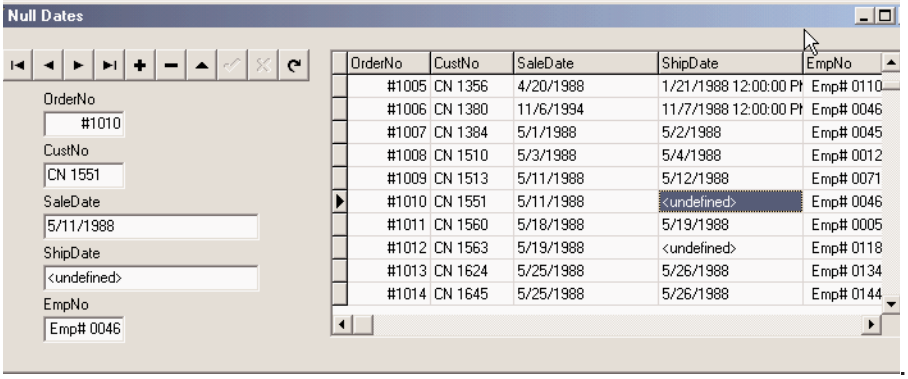
The NullDates program displays a specific text for dates that have a null value and clears the field (setting it to the null value) when the user uses an empty string in input. Here is the relevant code of the two event handlers for the field:

```
procedure TForm1.Table1ShipDateGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
```

```
    if Sender.IsNull then
      Text := '<undefined>'
    else
      Text := Sender.AsString;
  end;

  procedure TForm1.Table1ShipDateSetText(Sender: TField; const Text: String);
  begin
    if Text = '' then
      Sender.Clear
    else
      Sender.AsString := Text;
  end;
```

In Figure 13.11 you can see an example of the output of this program, with undefined (or null) values for some shipping dates.

FIGURE 13.11: By handling the OnGetText and OnSetText events of a date field, the NullDates example displays a specific output for null values


Null Dates

OrderNo: #1010
CustNo: CN 1551
SaleDate: 5/11/1988
ShipDate: <undefined>
EmpNo: Emp# 0046

OrderNo	CustNo	SaleDate	ShipDate	EmpNo
#1005	CN 1356	4/20/1988	1/21/1988 12:00:00 P	Emp# 0110
#1006	CN 1380	11/6/1994	11/7/1988 12:00:00 P	Emp# 0046
#1007	CN 1384	5/1/1988	5/2/1988	Emp# 0045
#1008	CN 1510	5/3/1988	5/4/1988	Emp# 0012
#1009	CN 1513	5/11/1988	5/12/1988	Emp# 0071
#1010	CN 1551	5/11/1988	<undefined>	Emp# 0046
#1011	CN 1560	5/18/1988	5/19/1988	Emp# 0005
#1012	CN 1563	5/19/1988	<undefined>	Emp# 0118
#1013	CN 1624	5/25/1988	5/26/1988	Emp# 0134
#1014	CN 1645	5/25/1988	5/26/1988	Emp# 0144

Navigating a Dataset

We've seen that a dataset has only one active record, and you can imagine that the active record changes often, in response of user actions or because of internal commands given to the dataset. To move around the dataset and change the active record, there are methods of the TDataSet class, as you can see in Listing 13.1, particularly in the section commented as "position, movement." You can move to the next or previous record, jump back and forth by a given number of records (with MoveBy), or go directly to the first or last record of the dataset. These operations of the dataset are generally available in the DBNavigator component or in the standard dataset actions, and they are not particularly complex to understand.

What is not obvious, though, is how a dataset handles the extreme positions. If you open any dataset with a navigator attached, you can see that as you move on record by record, the Next button remains enabled even when you've reached the last record. It's only when you try to move forward after the last record that the current record apparently doesn't change and the button is disabled. This is because the Eof test (end of file) succeeds only when the cursor has been moved to a special position after the last record. If you jump to the end with the Last button, instead, you'll immediately be at the very end. You'll see exactly the same behavior for the first record (and the Bof test). As we'll see in a while, this approach is very handy, as we can scan a dataset testing for Eof to be True and, at this point, we know we've also already processed the last record of the dataset.

NOTE Handling this special record positions before the beginning and after the end of the dataset, which are called *cracks*, is very important (and quite confusing) when you write a custom dataset, as we'll see in Chapter 18.

Besides moving around record by record or by a given number of records, programs might need to jump to specific records or positions. Some datasets support the RecordCount property and allow movement to a record at a given position in the dataset using the RecNo property. These properties can be used only for datasets that support positions natively, which basically excludes all client/server architectures, unless you grab all of the records in a local cache (something you'll generally want to avoid) and then navigate on the cache. As we'll see in the next chapter, when you open a query on a SQL server you fetch only the records you are using, so Delphi doesn't know the record count, at least not in advance.

There are two alternatives you can use to refer to a record in a dataset, regardless of its type:

- You can save a reference to the current record and then jump back to it after moving around. This is accomplished by using bookmarks, either in the TBookmark or the more modern TBookmarkStr form. This approach is discussed in the upcoming section "Using Bookmarks."
- You can locate a record of the dataset matching given criteria, using the Locate method. This even works after you close and reopen the dataset, because you're working at a logical (and not physical) level. This approach is presented in the next section.

Locating Records in a Table

To show you an example of the use of the Locate method, I've built the Search example, which has a table connected to EMPLOYEE.DB. The form I've prepared has the data-aware edit boxes inside a scroll box aligned to the client area, so that a user can freely resize the form without any problems. When the form becomes too small, scroll bars will appear automatically in the area holding the edit boxes. Another feature is a toolbar with buttons connected to some of the predefined dataset actions available in the ActionList component plus two custom actions to host the search code.

The searching capabilities are activated by the two buttons connected to custom actions. The first button is connected to `ActionGoto`, used for an exact match, and the second to `ActionGoNear`, for a partial match. In both cases, we want to compare the text in the edit box with the `LastName` fields of the EMPLOYEE table. If the local table has an index on the field (as in the specific case) `Locate` will use it, but the method will work with or without indexes (only at a different speed).

If you've never used `Locate`, at first sight the help file won't be terribly clear. The idea is that you must provide a list of fields you want to search, and a list of values, one for each field. If you pass only one field, the value is passed directly, as in the case of the example:

```
procedure TSearchForm.ActionGotoExecute(Sender: TObject);
begin
  if not Table1.Locate ('LastName', EditName.Text, []) then
    MessageDlg ('"' + EditName.Text + '" not found', mtError, [mbOk], 0);
end;
```

If you search for multiple fields, you have to pass a variant array with the list of the values you want to match. The variant array can be created from a constant array with the `VarArrayOf` function or from scratch using the `VarArrayCreate` call. This is a code snippet from the example:

```
Table1.Locate ('LastName;FirstName', VarArrayOf (['Cook', 'Kevin']), [])
```

Finally, we can use the same method to look for a record even if we know only the initial portion of the field we are looking for. Simply add the `loPartialKey` flag to the `Options` parameter (the third) of the `Locate` call.

NOTE

Using `Locate` makes sense specifically for local tables, but doesn't port very well to client/server applications. In fact, on local tables this technique is rather optimized by letting the dataset read in only the records it is looking for, using local indexes. On a SQL server, instead, similar client-side techniques imply moving all the data to the client application first, which is generally a bad idea. Locating the data should be performed with restricted SQL statements. You can still call `Locate` after you're retrieved a limited dataset. For example, you can search a customer by name after you've selected all the customer of a given town or area, obtaining a result set of a limited size. There's more on this topic in the next chapter, which is devoted to client/server development.

The Total of a Table Column

So far in our examples, the user can view the current contents of a database table and manually edit the data or insert new records. Now we will see how we can change some data in the table through the program code. The idea behind this example is quite simple. The EMPLOYEE table we have been using has a Salary field. A manager of the company could indeed browse through the table and change the salary of a single employee. But what will be the total salary expense for the company? And what if the manager wants to give a 10 percent salary increase (or decrease) to everyone?

These are the two aims of the Total example, which is an extension of the previous program. The toolbar of this new example has some more buttons and actions. There are a few other minor changes from the previous example. I opened the Fields editor of the table and removed the Table1Salary field, which was defined as a TFloatField. Then I selected the New Field command and added the same field, with the same name, but using the TCurrencyField data type. This is not a calculated field; it's simply a field converted into a new (but equivalent) data type. Using this new field type the program will default to a new output format, suitable for currency values.

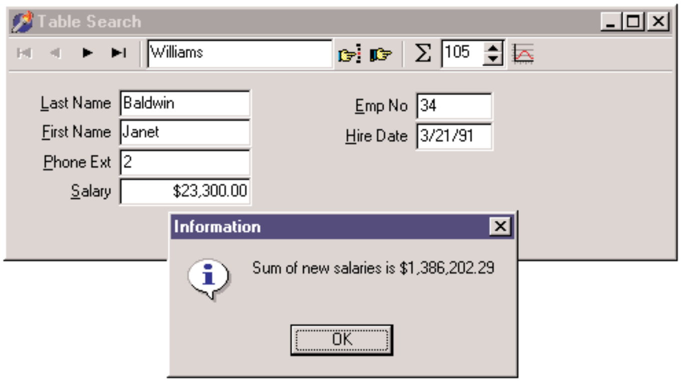
Now we can turn our attention to the code of this new program. First, let's look at the code of the total action. This action lets you calculate the sum of the salaries of all the employees, then edit some of the values, and compute a new total. Basically, we need to scan the table, reading the value of the Table1Salary field for each record:

```
var
  Total: Real;
begin
  Total := 0;
  Table1.First;
  while not Table1.EOF do
  begin
    Total := Total + Table1Salary.Value;
    Table1.Next;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOk], 0);
end
```

This code works, as you can see from the output in Figure 13.12, but it has some problems. One problem is that the record pointer is moved to the last record, so the previous position in the table is lost. Another is that the user interface is refreshed many times during the operation.

FIGURE 13.12:

The output of the Total program, showing the total salaries of the employees



Sum of new salaries is $1,386,202.29

Last Name: Baldwin
First Name: Janet
Phone Ext: 2
Salary: $23,300.00
Emp No: 34
Hire Date: 3/21/91

Using Bookmarks

To avoid these two problems, we need to disable updates and to store the current position of the record pointer in the table and restore it at the end. This can be accomplished using a *table bookmark*, a special variable storing the position of a record in a database table. Delphi's traditional approach is to declare a variable of the TBookmark data type, and initialize it while getting the current position from the table:

```
var
  Bookmark: TBookmark;
begin
  Bookmark := Table1.GetBookmark;
```

At the end of the ActionTotalExecute method, we can restore the position and delete the bookmark with the following two statements:

```
Table1.GotoBookmark (Bookmark);
Table1.FreeBookmark (Bookmark);
```

As a better (and more up-to-date) alternative, we can use the Bookmark property of the TDataset class, which refers to a bookmark that is disposed of automatically. (This is technically implemented as an *opaque string*, a structure subject to string lifetime management, but it is not a string, so you're not supposed to look at what's inside it.) This is how you can modify the code above:

```
var
  Bookmark: TBookmarkStr;
begin
  Bookmark := Table1.Bookmark;
  ...
  Table1.Bookmark := Bookmark;
```

To avoid the other side effect of the program (we see the records scrolling while the routine browses through the data), we can temporarily disable the visual controls connected with the table. The table has a DisableControls method we can call before the while loop starts and an EnableControls method we can call at the end, after the record pointer is restored.

TIP Disabling the data-aware controls connected with a table during long operations not only improves the user interface (since the output is not changing constantly), it also speeds up the program considerably. In fact, the time spent to update the user interface is much greater than the time spent performing the calculations. To test this, try commenting out the DisableControls and EnableControls methods of the Total example, and see the speed difference.

Finally, we face some dangers from errors in reading the table data, particularly if the program is reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. So we should use a try/finally block. Actually, if you want to make the program 100 percent error-proof, you should use two nested try/finally blocks. Including this change and the two discussed above, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);
var
  Bookmark: TBookmarkStr;
  Total: Real;
begin
  Bookmark := Table1.Bookmark;
  try
    Table1.DisableControls;
    Total := 0;
    try
      Table1.First;
      while not Table1.EOF do
      begin
        Total := Total + Table1Salary.Value;
        Table1.Next;
      end;
    finally
      Table1.EnableControls;
  end
  finally
    Table1.Bookmark := Bookmark;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

NOTE I've written this code to show you an example of a loop to browse the contents of a table, but keep in mind that there is an alternative approach based on the use of a SQL query returning the sum of the values of a field. When you use a SQL server, the speed advantage of a SQL call to compute the total can be very large, since you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result.

Editing a Table Column

The code of the increase action is similar to the one we have just seen. The `ActionIncreaseExecute` method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference. When you increase the salary, you actually change the data in the table. The two key statements are within the `while` loop:

```
while not Table1.EOF do
begin
  Table1.Edit;
  Table1Salary.Value := Round (Table1Salary.Value * SpinEdit1.Value) / 100;
  Total := Total + Table1Salary.Value;
  Table1.Next;
end;
```

The first statement brings the table into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary, multiplying the old one by the value of the SpinEdit component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount—even double the salary of each employee—with the click of a button.

WARNING Notice that the table enters the edit mode every time the `while` loop is executed. This is because in a dataset, edit operations can take place only one record at a time. You must finish the edit operation, calling `Post` or moving to a different record as in the code above. At that time, if you want to change another record, you have to enter edit mode once more.

Customizing a Database Grid

Unlike most other data-aware controls, which are quite simple to use, the DBGrid control has many options and is more powerful than you might think. The following sections explore some of the advanced operations you can do using a DBGrid control. A first example shows how to draw in a grid, a second one shows how to clone the behavior of a check box for a

Boolean selection inside a grid, and the final example shows how to use the multiple-selection feature of the grid.

Painting a DBGrid

There are many reasons you might want to customize the output of a grid. A good example is to highlight specific fields or records. Another is to provide some form of output for fields that usually don't show up in the grid, such as BLOB, graphic, and memo fields.

To thoroughly customize the drawing of a DBGrid control, you have to set its `DefaultDrawing` property to False and handle its `OnDrawColumnCell` event. In fact, if you leave the value of `DefaultDrawing` set to True, the grid will display the default output before the method is called. This way, all you can do is add something to the default output of the grid, unless you decide to draw over it, which will take extra time and cause flickering.

The alternative approach is to call the `DefaultDrawColumnCell` method of the grid, perhaps after changing the current font or restricting the output rectangle. In this last case you can provide an extra drawing in a cell and let the grid fill the remaining area with the standard output. This is what I've done in the DrawData program.

The DBGrid control in this example, which is connected to the commonly used `BIOLIFE` table of the DBDEMOS database, has the following properties:

```
object DBGrid1: TDBGrid
  Align = alClient
  DataSource = DataSource1
  DefaultDrawing = False
  Font.Height = -16
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  TitleFont.Height = -11
  TitleFont.Name = 'MS Sans Serif'
  TitleFont.Style = []
  OnDrawColumnCell = DBGrid1DrawColumnCell
end
```

The `OnDrawColumnCell` event handler is called once for every cell of the grid and has several parameters, including the rectangle corresponding to the cell, the index of the column we have to draw, the column itself (with the field, its alignment, and other subproperties), and the status of the cell. How can we set the color of specific cells to red? We can simply change it in the special cases:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
```

```
    // red font color if length > 100
    if (Column.Field = Table1Lengthcm) and (Table1Lengthcm.AsInteger > 100) then
      DBGrid1.Canvas.Font.Color := clRed;

    // default drawing
    DBGrid1.DefaultDrawDataCell (Rect, Column.Field, State);
  end;
```

The next step is to draw the memo and the graphic fields. For the memo we can simply implement the memo field's OnGetText and OnSetText events. In fact, the grid will even allow editing on a memo field if its OnSetText event is not nil. Here is the code of the two event handlers. I've used Trim to remove trailing nonprinting characters, which make the text appear to be empty when editing:

```
procedure TForm1.Table1NotesGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := Trim (Sender.AsString);
end;

procedure TForm1.Table1NotesSetText(Sender: TField; const Text: String);
begin
  Sender.AsString := Text;
end;
```

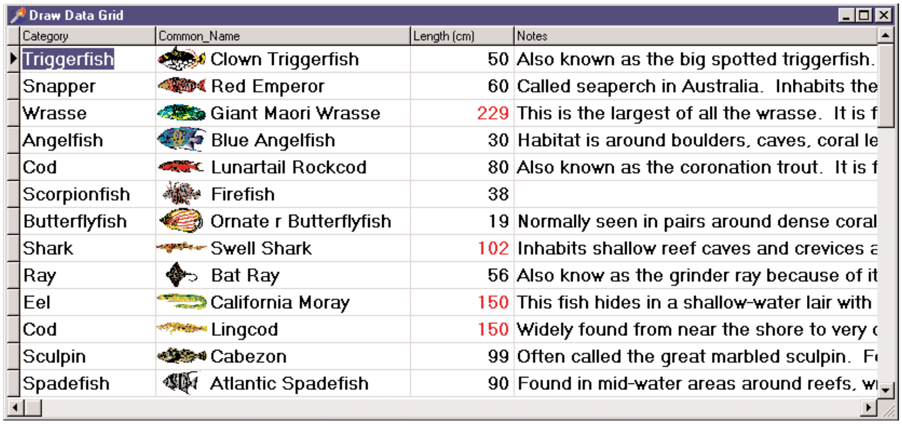
For the image, the simplest approach is to create a temporary TBitmap object, assign the graphics field to it, and paint the bitmap to the Canvas of the grid. As an alternative, I've removed the graphic field from the grid, by setting its Visible property to False, and added the image to the fish name, with the following extra code in the OnDrawColumnCell event handler:

```
var
  Bmp: TBitmap;
  OutRect: TRect;
  BmpWidth: Integer;
begin
  // default output rectangle
  OutRect := Rect;

  if Column.Field = Table1Common_Name then
  begin
    // draw the image
    Bmp := TBitmap.Create;
    try
      Bmp.Assign (Table1Graphic);
      BmpWidth := (Rect.Bottom - Rect.Top) * 2;
      OutRect.Right := Rect.Left + BmpWidth;
    DBGrid1.Canvas.StretchDraw (OutRect, Bmp);
  finally
    Bmp.Free;
  end;
  // reset output rectangle, leaving space for the graphic
  OutRect := Rect;
  OutRect.Left := OutRect.Left + BmpWidth;
end;

// red font color if length > 100 (omitted - see above)

// default drawing
DBGrid1.DefaultDrawDataCell (OutRect, Column.Field, State);
```

As you can see in the code above, the program shows the image in a small rectangle on the left of the grid cell and then changes the output rectangle to the remaining area before activating the default drawing. You can see the effect in Figure 13.13.

FIGURE 13.13:

The DrawData program displays a grid that includes the text of a memo field and the ubiquitous Borland fishes.



Category	Common_Name	Length (cm)	Notes
Triggerfish	Clown Triggerfish	50	Also known as the big spotted triggerfish.
Snapper	Red Emperor	60	Called seaperch in Australia. Inhabits the
Wrasse	Giant Maori Wrasse	229	This is the largest of all the wrasse. It is f
Angelfish	Blue Angelfish	30	Habitat is around boulders, caves, coral le
Cod	Lunartail Rockcod	80	Also known as the coronation trout. It is f
Scorpionfish	Firefish	38	
Butterflyfish	Ornate r Butterflyfish	19	Normally seen in pairs around dense coral
Shark	Swell Shark	102	Inhabits shallow reef caves and crevices a
Ray	Bat Ray	56	Also know as the grinder ray because of it
Eel	California Moray	150	This fish hides in a shallow-water lair with
Cod	Lingcod	150	Widely found from near the shore to very c
Sculpin	Cabezon	99	Often called the great marbled sculpin. F
Spadefish	Atlantic Spadefish	90	Found in mid-water areas around reefs, wi

A Check Box Cell

Another common extension of the DBGrid control, found in many third-party components, is the use of check boxes to select the status of Boolean field values. A simple way to do this is to place a DBCheckBox control in front of the grid when the user selects the corresponding item. I've done this in the CheckDbg example, which uses the Workers table created in the DbAware example discussed earlier in this chapter.

The form displayed by the program contains only the grid and the check box. This is a summary of the textual description of the form:

```
object DbaForm: TDbaForm
  OnCreate = FormCreate
  object DBGrid1: TDBGrid
```

```
      Align = alClient
      DataSource = DataSource1
      OnColEnter = DBGrid1ColEnter
      OnDrawColumnCell = DBGrid1DrawColumnCell
      OnKeyPress = DBGrid1KeyPress
    end
    object DBCheckBox1: TDBCheckBox
      Caption = 'Senior'
      DataField = 'Senior'
      DataSource = DataSource1
      ValueChecked = 'True'
      ValueUnchecked = 'False'
      Visible = False
    end
    object Table1: TTable
      DatabaseName = 'DBDEMOS'
      TableName = 'Workers'
    end
  end
```

Notice that the check box is initially hidden and that the program handles several events of the DBGrid control. The first is the OnDrawColumnCell event, which is not used to customize the drawing (the DefaultDrawing property is set to True), but only to compute the position of the check box when a cell of the corresponding field is selected:

```
procedure TDbaForm.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  if (gdFocused in State) and (Column.Field = Table1Senior) then
  begin
    DBCheckBox1.SetBounds (
      Rect.Left + DBGrid1.Left + 1,
      Rect.Top + DBGrid1.Top + 1,
      Rect.Right - Rect.Left + 1,
      Rect.Bottom - Rect.Top + 1);
  end;
end;
```

The check box itself is displayed or hidden as the user enters or exits the corresponding column, by the handler of the OnColEnter event. Note that we cannot refer to the column by position, since a user can move the columns:

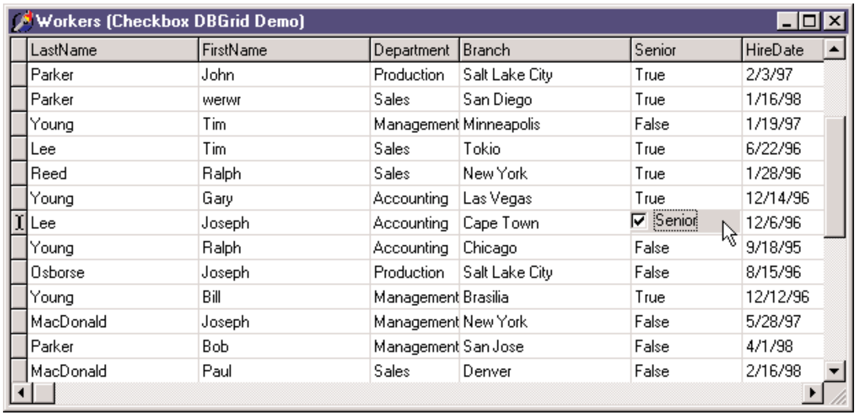
```
procedure TDbaForm.DBGrid1ColEnter(Sender: TObject);
begin
  if DBGrid1.Columns [DBGrid1.SelectedIndex].Field = Table1Senior then
    DBCheckBox1.Visible := True
  else
    DBCheckBox1.Visible := False;
end;
```

Finally, as an extra extension, when the check box is visible (that is, when the user has activated the corresponding field), the program intercepts the keyboard input in the grid, toggling the selection of the check box instead of accepting the input:

```
procedure TDbaForm.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
  if DBCheckBox1.Visible and (Ord (Key) > 31) then
  begin
    Key := #0;
    Table1.Edit;
    DBCheckBox1.Checked := not DBCheckBox1.Checked;
    DBCheckBox1.Field.AsBoolean := DBCheckBox1.Checked;
  end;
end;
```

To make this work we must not only toggle the status of the check box, but also go into edit mode and update the data of the field. You can see an example of the output of this program in Figure 13.14.

FIGURE 13.14: The grid of the CheckDbg example uses a check box for selecting the value of a Boolean field.



Workers (Checkbox DBGrid Demo)

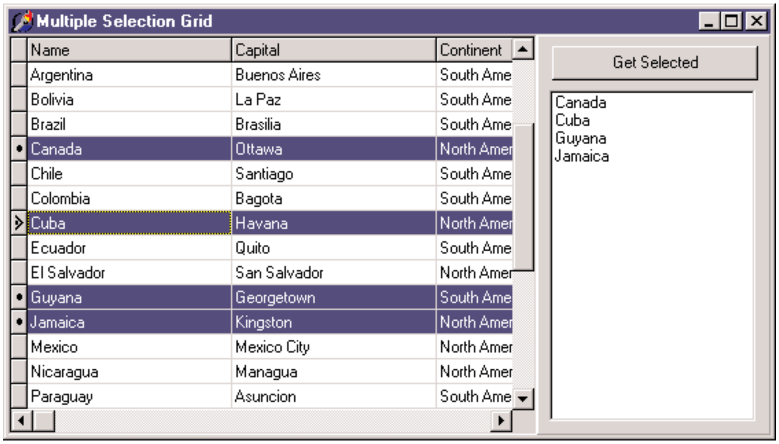
LastName	FirstName	Department	Branch	Senior	HireDate
Parker	John	Production	Salt Lake City	True	2/3/97
Parker	werwr	Sales	San Diego	True	1/16/98
Young	Tim	Management	Minneapolis	False	1/19/97
Lee	Tim	Sales	Tokio	True	6/22/96
Reed	Ralph	Sales	New York	True	1/28/96
Young	Gary	Accounting	Las Vegas	True	12/14/96
Lee	Joseph	Accounting	Cape Town	☑ Senior	12/6/96
Young	Ralph	Accounting	Chicago	False	9/18/95
Osborse	Joseph	Production	Salt Lake City	False	8/15/96
Young	Bill	Management	Brasilia	True	12/12/96
MacDonald	Joseph	Management	New York	False	5/28/97
Parker	Bob	Management	San Jose	False	4/1/98
MacDonald	Paul	Sales	Denver	False	2/16/98

A Grid Allowing Multiple Selection

The third and last example of customizing the DBGrid control relates to multiple selection. You can set up the DBGrid so that a user can select multiple rows (that is, multiple records). This is very easy, since all you have to do is toggle the `dgMultiSelect` element of the `Options` property of the grid. Once you've selected this option, a user can keep the Ctrl key pressed and click with the mouse to select multiple rows of the grid, with the effect you can see in Figure 13.15.

FIGURE 13.15:

The MltGrid example has a DBGrid control that allows the selection of multiple rows.



	Name	Capital	Continent
	Argentina	Buenos Aires	South Ame
	Bolivia	La Paz	South Ame
	Brazil	Brasilia	South Ame
●	Canada	Ottawa	North Amer
	Chile	Santiago	South Ame
	Colombia	Bagota	South Ame
▶	Cuba	Havana	North Amer
	Ecuador	Quito	South Ame
	El Salvador	San Salvador	North Amer
●	Guyana	Georgetown	South Ame
●	Jamaica	Kingston	North Amer
	Mexico	Mexico City	North Amer
	Nicaragua	Managua	North Amer
	Paraguay	Asuncion	South Ame

Canada
Cuba
Guyana
Jamaica

Since the database table can have only one active record, what information is stored in the grid for the selected items? The grid simply keeps a list of bookmarks to the selected records. This list is available in the `SelectedRows` property, which is of type `TBookmarkList`. Besides accessing the number of objects in the list with the `Count` property, you can get to each bookmark with the `Items` property, which is the default array property. Each item of the list is on a `TBookmarkStr` type, which represents a bookmark pointer you can assign to the `Bookmark` property of the table.

NOTE The `TBookmarkStr` is a string type for convenience, but its data should be considered "opaque" and volatile. You shouldn't rely on any particular structure to the data you may find if you peek at a bookmark's value, and you shouldn't hold on to the data too long or store it in a separate file. Bookmark data will vary with database driver and index configuration, and it may be rendered unusable when rows are added to or deleted from the dataset (by you or by other users of the database).

To summarize the steps, here is the code of the MltGrid example, activated by pressing the button to move the Name field of the selected records to the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  BookmarkList: TBookmarkList;
  Bookmark: TBookmarkStr;
begin
  // store the current position
  Bookmark := Table1.Bookmark;
  try
    // empty the list box
    ListBox1.Items.Clear;
    // get the selected rows of the grid
    BookmarkList := DbGrid1.SelectedRows;
    for I := 0 to BookmarkList.Count - 1 do
    begin
      // for each, move the table to that record
      Table1.Bookmark := BookmarkList[I];
      // add the name field to the listbox
      ListBox1.Items.Add (Table1.FieldByName ('Name').AsString);
    end;
  finally
    // go back to the initial record
    Table1.Bookmark := Bookmark;
  end;
end;
```

Dragging to a Grid

Another interesting technique is to use dragging with grids. Dragging *from* a grid is not particularly difficult, as you know which are the current record and the column the user has selected. Dragging *to* a grid, instead, is apparently hard to program. You might remember that in Chapter 3, "The Object Pascal Language: Inheritance and Polymorphism," I mentioned the "protected hack;" this is the technique I'm going to use to implement dragging to a grid.

The example, called DragToGrid, has a grid connected to the COUNTRY.DB demo table, an edit where you can type the new value for a field, and a label you can drag over a cell of the grid to modify the related field. The real problem is how to determine this field. The code is only a few lines, as you can see below, but it is certainly cryptic, and requires some explanation:

```
type
  TDBGHack = class (TDbGrid)
  end;

procedure TFormDrag.DBGrid1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  gc: TGridCoord;
begin
  gc := TDBGHack (DbGrid1).MouseCoord (x, y);
  if (gc.y > 0) and (gc.x > 0) then
  begin
    DbGrid1.DataSource.DataSet.MoveBy (gc.y - TDBGHack(DbGrid1).Row);
    DbGrid1.DataSource.DataSet.Edit;
    DBGrid1.Columns.Items [gc.X - 1].Field.AsString := EditDrag.Text;
  end;
  DBGrid1.SetFocus;
end;
```

The first operation is to determine the cell over which the mouse was released. Starting with the `X` and `Y` mouse coordinates, we can call the protected `MouseCoord` method to access the row and column of the cell. Unless the drag target is the very first row (usually hosting the titles) or the first column (usually hosting the indicator), the program moves the current record by the difference between the requested row (`gc.y`) and the current active row (the protected `Row` property of the grid). The next step is to put the dataset into edit mode, grab the field of the target column (`Columns.Items [gc.X - 1].Field`), and change its text.

Database Applications with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this is certainly not required. When you need to have very precise control over the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. My personal view is that this is necessary only in very specific cases, as you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you understand the default behavior of Delphi, and it will help me introduce some more database-related events (discussed in the sections "Database Events" and "Field Events").

The development of an application not based on data-aware controls can follow two different approaches. You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a much more customized approach. I'll demonstrate the first technique in the NonAware example and the latter in the SendToDb example.

Mimicking Delphi Data-Aware Controls

If you want to build an application that doesn't use data-aware controls but behaves like a standard Delphi application, you can simply write event handlers for the operations that would be performed automatically by data-aware controls. Basically you need to place the dataset in edit mode as the user changes the content of the visual controls, and update the field objects of the dataset as the user exits from the controls, moving the focus to another element.

TIP This approach can be handy for integrating a control that's not data-aware into a standard application. A good example is the use of the DateTimePicker control for selecting a date, as demonstrated later in the section "Editing Dates with a Calendar."

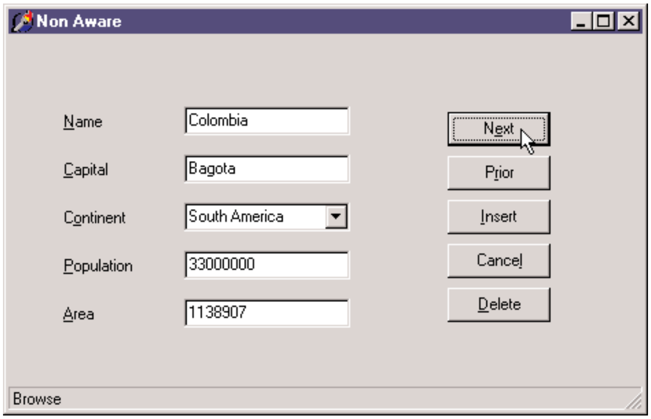
The other element of the NonAware example is a list of buttons corresponding to some of those in the DBNavigator control and connected to five custom actions. I cannot use the standard dataset actions for this example simply because they automatically hook to the data source associated with the control having the focus, a mechanism that fails with the non–data-aware edit boxes of this example.

The program has several event handlers we've not used for past applications using data-aware controls. First of all, we have to show the data of the current record in the visual controls (as in Figure 13.16), by handling the `OnDataChange` event of the `DataSource1` component:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  EditName.Text := Table1Name.AsString;
  EditCapital.Text := Table1Capital.AsString;
  ComboContinent.Text := Table1Continent.AsString;
  EditArea.Text := Table1Area.AsString;
  EditPopulation.Text := Table1Population.AsString;
end;
```

FIGURE 13.16:

The output of the Non-Aware example in Browse mode. The program manually fetches the data every time the current record changes.



Non Aware

Name	Colombia
Capital	Bagota
Continent	South America
Population	33000000
Area	1138907

Browse

The handler of the `OnStateChange` event of the control displays the status of the table in a status bar control. As the user starts typing in one of the edit boxes or drops down the combo box list, the program sets the table in edit mode:

```
procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
  if not (Table1.State in [dsEdit, dsInsert]) then
    Table1.Edit;
end;
```

This method is connected with the `OnKeyPress` event of the five components and is similar to the `OnDropDown` event handler of the combo box. As the user leaves one of the visual controls, the handler of the OnExit event copies the data to the corresponding field, as in this case:

```
procedure TForm1.EditCapitalExit(Sender: TObject);
begin
  if (Table1.State in [dsEdit, dsInsert]) then
    Table1Capital.AsString := EditCapital.Text;
end;
```

The operation takes place only if the table is in edit mode; that is, only if the user has typed in this or another control. This is not really ideal, because extra operations are done even if the text of the edit box didn't change, but the extra steps happen fast enough not to be a concern. For the first edit box, we check the text before copying it, raising an exception if the edit box is empty:

```
procedure TForm1.EditNameExit(Sender: TObject);
begin
  if (Table1.State in [dsEdit, dsInsert])then
    if EditName.Text <> '' then
      Table1Name.AsString := EditName.Text
    else
    begin
      EditName.SetFocus;
      raise Exception.Create ('Undefined Country');
    end;
end;
```

An alternative approach for testing the value of a field is to handle the BeforePost event of the dataset. Keep in mind that in this example, the posting operation is not handled by a specific button but takes place as soon as a user moves to a new record or inserts a new one:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
  if Table1Area.Value < 100 then
    raise Exception.Create ('Area too small');
end;
```

In each of these cases, an alternative to raising an exception is to set a default value. However, if a field has a default value it is better to set it up front, so that a user can see which value will be sent to the database. To accomplish this, you can handle the AfterInsert event of a dataset, which is fired immediately after a new record has been created (we could have used the OnNewRecord event, as well):

```
procedure TForm1.Table1AfterInsert(DataSet: TDataSet);
begin
  Table1Continent.Value := 'Asia';
end;
```

Sending Requests to the Database

You can further customize the user interface of your application if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This allows you complete freedom, although there might be some side effects (such as limited ability to handle concurrency, something I'll discuss in the next chapter).

For this new example, I've replaced the first edit box with another combo box, and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom ones, used to get the data from the database and send an update to it. To underline the difference of this example, I've even removed the DataSource component.

The `GetData` method, connected with the corresponding button, simply gets the fields corresponding to the record indicated in the first combo box:

```
procedure TForm1.GetData;
begin
  Table1.FindNearest ([ComboName.Text]);
  ComboName.Text := Table1Name.AsString;
  EditCapital.Text := Table1Capital.AsString;
  ComboContinent.Text := Table1Continent.AsString;
  EditArea.Text := Table1Area.AsString;
  EditPopulation.Text := Table1Population.AsString;
end;
```

This method is called whenever the user presses the button, selects an item of the combo box, or presses the Enter key while in the combo box:

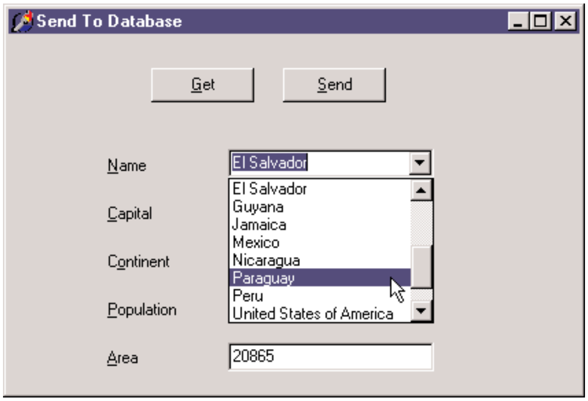
```
procedure TForm1.ComboNameClick(Sender: TObject);
begin
  GetData;
end;

procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
    GetData;
end;
```

To make this example work smoothly, at start-up the combo box is filled with all the names of the countries of the table:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // fill the list of names
  Table1.Open;
  while not Table1.Eof do
  begin
```

```
      ComboName.Items.Add (Table1Name.AsString);
      Table1.Next;
    end;
  end;
```

With this approach, the combo box becomes a sort of selector of the record, as you can see in Figure 13.17. Notice that thanks to this selection, the program doesn't need navigational buttons.



Name: El Salvador

El Salvador
Guyana
Jamaica
Mexico
Nicaragua
Paraguay
Peru
United States of America

Area: 20865

FIGURE 13.17: In the SendToDb example, you can select the record you want to see in a combo box.

Finally, the user can change the values of the controls and click the Send button. The code to be executed depends on whether the operation is an update or an insert. We can determine this by looking at the name (although with this code, a wrong name cannot be modified any more):

```
procedure TForm1.SendData;
begin
  // raise an exception if there is no name
  if ComboName.Text = '' then
    raise Exception.Create ('Insert the name');

  // check if the record is already in the table
  if Table1.FindKey ([ComboName.Text]) then
  begin
    // modify found record
    Table1.Edit;
    Table1Capital.AsString := EditCapital.Text;
    Table1Continent.AsString := ComboContinent.Text;
    Table1Area.AsString := EditArea.Text;
    Table1Population.AsString := EditPopulation.Text;
    Table1.Post;
  end
  else
```

```
begin
  // insert new record
  Table1.InsertRecord ([ComboName.Text, EditCapital.Text,
    ComboContinent.Text, EditArea.Text, EditPopulation.Text]);
  // add to list
  ComboName.Items.Add (ComboName.Text)
end;
```

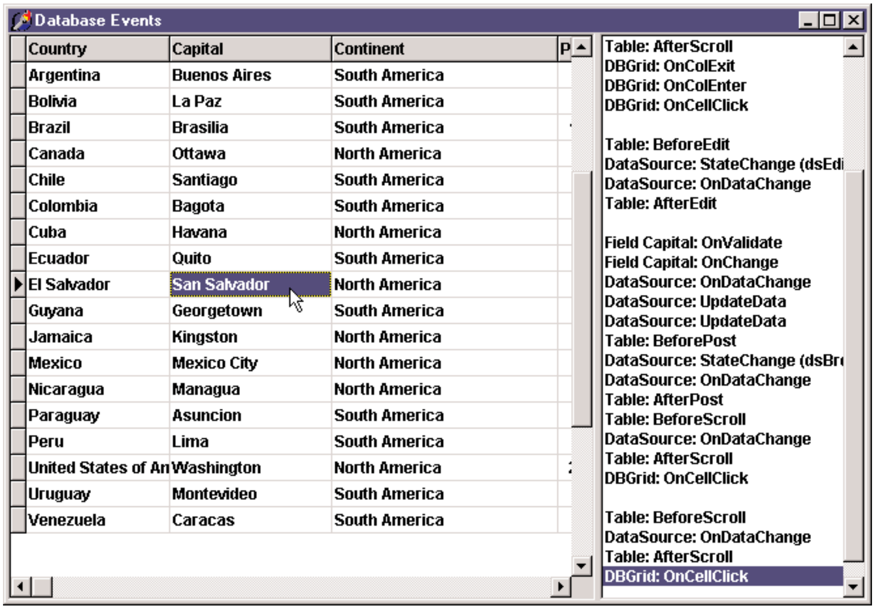
Before sending the data to the table, you can do any sort of validation test on the values. In this case, it doesn't make much sense to handle the events of the database components, because we have full control on when the update or insert operation is done.

Database Events

To further illustrate how you can use the events of a database application, I've written a simple program that logs all the events being fired. This program handles all of the events of a table and a data source component (although some of these events won't actually be executed, unless you add some extra code, as described later). For each event, I simply send its description to a list box, with the effect you can see in Figure 13.18.

FIGURE 13.18:

The output of the DbEvts program, which logs all the events related to database components



Country	Capital	Continent	P
Argentina	Buenos Aires	South America	
Bolivia	La Paz	South America	
Brazil	Brasilia	South America	
Canada	Ottawa	North America	
Chile	Santiago	South America	
Colombia	Bagota	South America	
Cuba	Havana	North America	
Ecuador	Quito	South America	
El Salvador	San Salvador	North America	
Guyana	Georgetown	South America	
Jamaica	Kingston	North America	
Mexico	Mexico City	North America	
Nicaragua	Managua	North America	
Paraguay	Asuncion	South America	
Peru	Lima	South America	
United States of An	Washington	North America	
Uruguay	Montevideo	South America	
Venezuela	Caracas	South America	

```
Table: AfterScroll
DBGrid: OnColExit
DBGrid: OnColEnter
DBGrid: OnCellClick

Table: BeforeEdit
DataSource: StateChange (dsEdi
DataSource: OnDataChange
Table: AfterEdit

Field Capital: OnValidate
Field Capital: OnChange
DataSource: OnDataChange
DataSource: UpdateData
DataSource: UpdateData
Table: BeforePost
DataSource: StateChange (dsBr
DataSource: OnDataChange
Table: AfterPost
Table: BeforeScroll
DataSource: OnDataChange
Table: AfterScroll
DBGrid: OnCellClick

Table: BeforeScroll
DataSource: OnDataChange
Table: AfterScroll
DBGrid: OnCellClick
```

Most of the event handlers simply display the name of the component and that of the event, as in

```
procedure TForm1.Table1AfterEdit(DataSet: TDataset);
begin
  AddToList ('Table: AfterEdit');
end;
```

The field events are slightly more complex, but they use a single handler for the various field components:

```
procedure TForm1.FieldChange(Sender: TField);
begin
  AddToList ('Field ' + Sender.FieldName + ': OnChange');
end;
```

The form's AddToList method adds a new item to the list box and selects it, automatically scrolling the list if required:

```
procedure TForm1.AddToList(Str: string);
begin
  // add item and select it
  Listbox1.ItemIndex := Listbox1.Items.Add (Str);
end;
```

Finally, the program has a pop-up menu connected to the list box to clear the list or save the items to a file. The menu also has a command you can use to add a blank line, thus separating blocks of events. This operation is also done automatically by a timer, which adds a blank line to the list box unless the last item is already an empty string. This makes the output more readable, as you can see in Figure 13.18.

It is very important to study the output of this program as well as its code. You can try doing all the various operations on the table using the DBGrid, such as inserting, editing, and deleting records, and see the corresponding effect in terms of events fired by the VCL components. To see even more events, you can set the Filtered property of the table to True, define a calculated field, try to cause errors (for example, by duplicating the value of the name field), add a check box to open or close the table, and so forth.

Field Events

The DbEvts program shows the calls to the OnChange and OnValidate events of the field objects. Two other events, OnSetText and OnGetText, are not shown, because the handlers of these events are not simply called to indicate that an operation occurred. On the contrary, their event handlers must perform the operation of getting data from or setting it to the corresponding field objects.

These two events are quite special, and their use is not as simple as it might seem at first sight. For this reason, they require a separate example, named FldText. This is only a slight revision of the DbAware example described earlier in this chapter, replacing the DBRadioGroup control with a DBListbox control. The problem is that a DBListBox control directly connects with a string field, while I want to connect it with an integer field, with each value indicating an option. Of course, I don't want a user to see or select a number, so I have to map the numbers stored in the database to the strings visible on the screen. In the earlier example, the DBRadioGroup control provided that mapping. Now I have to use an alternative approach.

In the FldText example, the Department field has two handlers for the `OnGetText` and `OnSetText` events. In the `OnGetText` event handler, you can extract the numeric value of the `Sender` field and set the value of the `Text` reference parameter:

```
procedure TDbaForm.Table1DepartmentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  case Sender.AsInteger of
    1: Text := 'Sales';
    2: Text := 'Accounting';
    3: Text := 'Production';
    4: Text := 'Management';
  else
    Text := '[Error]';
  end;
end;
```

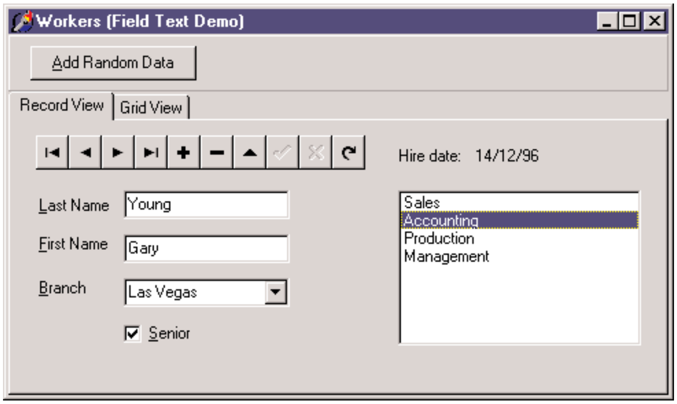
WARNING In the code of the `OnGetText` event handler you cannot refer to the text of the field, for example, using the `DisplayText` property or the `GetData` method, since they would call the `OnGetText` event, in an infinite recursion.

In the `OnSetText` event handler, you can examine the string and decide the value of the field according to the conversion rule, in this case a simple mapping of values done with an `if/then/else` statement:

```
procedure TDbaForm.Table1DepartmentSetText(Sender: TField; const Text: String);
begin
  if Text = 'Sales' then
    Sender.Value := 1
  else if Text = 'Accounting' then
    Sender.Value := 2
  else if Text = 'Production' then
    Sender.Value := 3
  else if Text = 'Management' then
    Sender.Value := 4
  else
    raise Exception.Create ('Error in Department field conversion');
end;
```

The effect is that not only is the value visible in the DBListBox (as you can see in Figure 13.19), it also shows up in the DBGrid. By contrast, in the DbAware example, the grid displayed the numeric value.

FIGURE 13.19:
The output of the FldText example, which demonstrates the use of the `OnGetText` and `OnSetText` events of the field objects



Hire date: 14/12/96

Last Name: Young

First Name: Gary

Branch: Las Vegas

☑ Senior

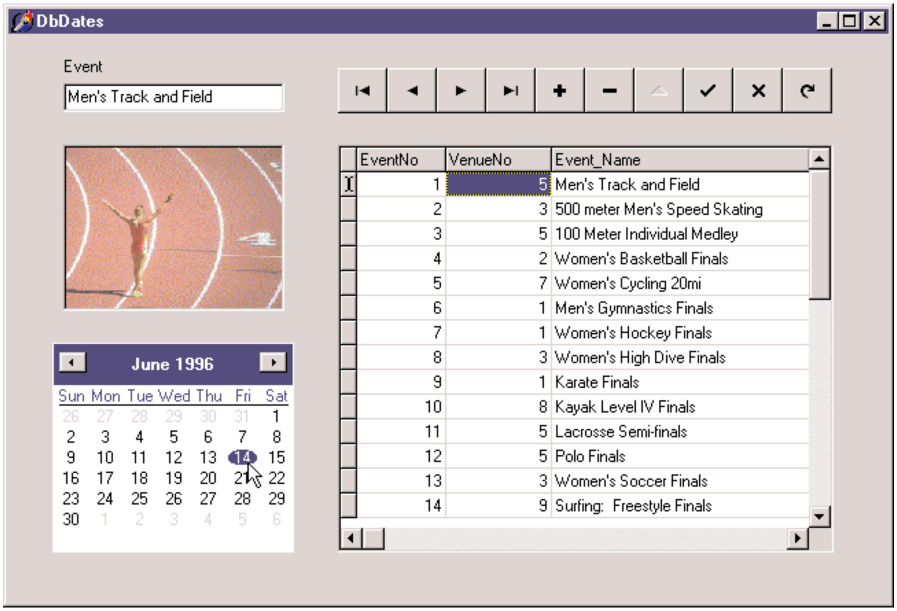
Sales
Accounting
Production
Management

Editing Dates with a Calendar

As a final example of the use of non–data-aware controls, the DbDates application shows how to use a MonthCalendar component to handle dates with a nice graphical component instead of a plain edit box. This example is based on the Events table from the DBDEMOS database, which lists Olympic events. The example uses (for the first time) a DBImage control, with the following settings (whose effect is illustrated in Figure 13.20):

```
object DBImage1: TDBImage
  DataField = 'Event_Photo'
  DataSource = DataSource1
  Stretch = True
end
```

NOTE Graphic, memo, and BLOB fields in Delphi are handled exactly like other fields. Just connect the proper editor or viewer, and most of the work is done behind the scenes by the system.



Event: Men's Track and Field

EventNo	VenueNo	Event_Name
1	5	Men's Track and Field
2	3	500 meter Men's Speed Skating
3	5	100 Meter Individual Medley
4	2	Women's Basketball Finals
5	7	Women's Cycling 20mi
6	1	Men's Gymnastics Finals
7	1	Women's Hockey Finals
8	3	Women's High Dive Finals
9	1	Karate Finals
10	8	Kayak Level IV Finals
11	5	Lacrosse Semi-finals
12	5	Polo Finals
13	3	Women's Soccer Finals
14	9	Surfing: Freestyle Finals

June 1996

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

FIGURE 13.20: The selection of a date with the monthly calendar

Although the DBImage control works with no extra effort on our part, we must connect the MonthCalendar control with the corresponding field by handling two events of the DataSource control:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  MonthCalendar1.Date := Table1Event_Date.Value;
end;

procedure TForm1.DataSource1UpdateData(Sender: TObject);
begin
  Table1Event_Date.Value := MonthCalendar1.Date;
end;
```

Besides copying the data back and forth, with the code listed above, the program must also put the table into edit mode as the user clicks the calendar control. The most obvious approach is to write a handler for the `OnClick` event of the control:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
  Table1.Edit;
end;
```

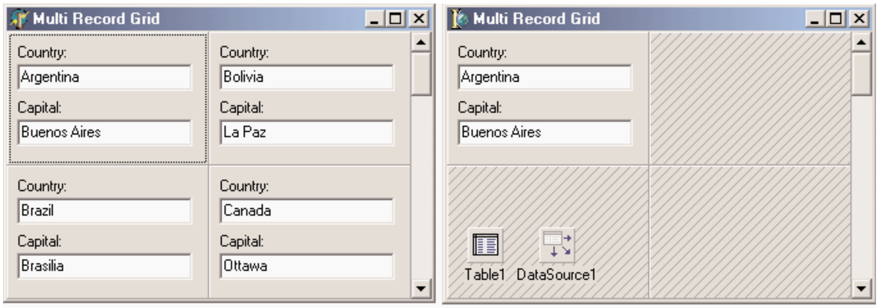
However, this code doesn't work properly. As you set the table in edit mode, the OnDataChange event is executed once more, resetting the selection in the calendar. The overall effect is that the user's first click doesn't change the selection. To avoid this problem we can set a flag in the OnClick event handler and test it in the OnDataChange event handler, or we can temporarily disconnect the second event handler. In the following code, I've taken the second approach:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
  // disconnect handler
  DataSource1.OnDataChange := nil;
  // set table in edit mode
  Table1.Edit;
  // reconnect handler
  DataSource1.OnDataChange := DataSource1DataChange;
end;
```

A Multirecord Grid

So far we have seen that you can either use a grid to display records of a database table or build a form with specific data-aware controls for the various fields, accessing the records one by one. There is a third alternative: use a multirecord object (a DBCtrlGrid), which allows you to place many data-aware controls in a small area of a form and automatically duplicate these controls for multiple records.

Here is what we can do to build the Multi1 example. Create a new blank form, place a Table component and a DataSource component in it, and connect them to the COUNTRY.DB table. Now place a DBCtrlGrid on the form, set its size and the number of rows and columns, and place two edit components connected with the Name and Capital fields of the table. To place these DBEdit components, you can also open the Fields editor and drag the two fields to the control grid. At design time, you simply work on the active portion of the grid (see Figure 13.21, on the right), and at run time, you can see these controls replicated multiple times (see Figure 13.21, on the left).



Country: Argentina
Capital: Buenos Aires

Country: Bolivia
Capital: La Paz

Country: Brazil
Capital: Brasilia

Country: Canada
Capital: Ottawa

Country: Argentina
Capital: Buenos Aires

FIGURE 13.21: The DBCtrlGrid of the Multi1 example at design time (on the right) and at run time (on the left)

You can simply set the number of columns and rows. Then each time you resize the control, the width and height of each panel are set accordingly. What is not available is a way to align the grid automatically to the client area of the form.

Moving Control Grid Panels

To improve the last example, we might resize the grid using the FormResize method. We could simply write the following code (in the Multi2 example):

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

This works, but it is not what I want. I'd like to increase the number of panels, not enlarge them. To accomplish this, we can define a minimum height for the panels and compute how many panels can fit in the available area each time the form is resized. For example, in Multi2, I've added one more statement to the FormResize method above, which now becomes

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.RowCount := (ClientHeight - Panel1.Height) div 100;
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

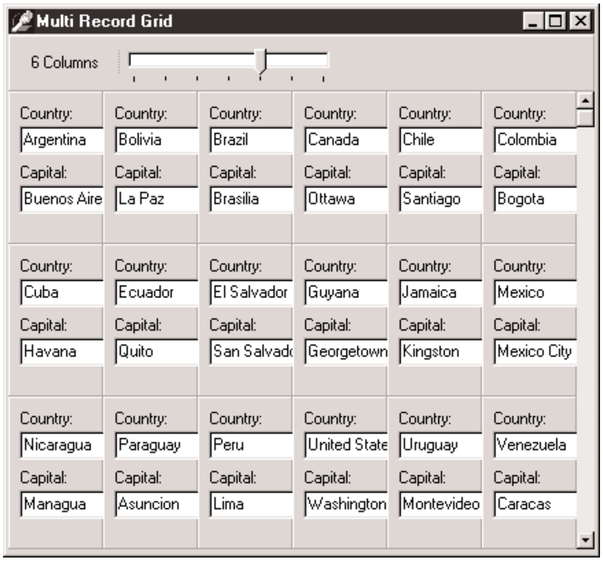
Instead of doing the same for the columns of the control grid component, I've added a TrackBar component to a panel. When the position of the trackbar changes (the range is from 2 to 10), the program sets the number of columns of the control grid and resizes it. In fact, if you simply set the number of columns, they'll have the same width as before. Here is the code of the trackbar's OnChange event handler:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  LabelCols.Caption := Format ('%d Columns', [TrackBar1.Position]);
  DBCtrlGrid1.ColCount := TrackBar1.Position;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

This code and the FormResize method above allow you to change the configuration of the control grid at run time in various ways. You can see an example of a crammed version of the form in Figure 13.22.

FIGURE 13.22:

The output of the Multi2 example, with an excessive number of columns



Multi Record Grid

6 Columns

Country:	Country:	Country:	Country:	Country:	Country:
Argentina	Bolivia	Brazil	Canada	Chile	Colombia
Capital:	Capital:	Capital:	Capital:	Capital:	Capital:
Buenos Aire	La Paz	Brasilia	Ottawa	Santiago	Bogota
Country:	Country:	Country:	Country:	Country:	Country:
Cuba	Ecuador	El Salvador	Guyana	Jamaica	Mexico
Capital:	Capital:	Capital:	Capital:	Capital:	Capital:
Havana	Quito	San Salvadc	Georgetown	Kingston	Mexico City
Country:	Country:	Country:	Country:	Country:	Country:
Nicaragua	Paraguay	Peru	United State	Uruguay	Venezuela
Capital:	Capital:	Capital:	Capital:	Capital:	Capital:
Managua	Asuncion	Lima	Washington	Montevideo	Caracas

Handling Database Errors

Another important element of database programming is handling database errors in custom ways. Of course, you can let Delphi show an exception message each time a database error occurs, but you might want to try to correct the errors or simply show more details. There are basically three approaches you can use to handle database-related errors:

- You can wrap a `try/except` block around risky database operations, such as a call to the `Open` method of a Query or to the `Post` method of a dataset. This is not possible when the operation is generated by the interaction with a data-aware control.
- You can install a handler for the `OnException` event of the global `Application` object or use the ApplicationEvents component, as described in the next example.
- You can handle specific events of the datasets related to errors, as `OnPostError`, `OnEditError`, `OnDeleteError`, and `OnUpdateError`. These events will be discussed later in the example.

While most of the exception classes in Delphi simply deliver an error message, with database exceptions you see a list of errors, showing local BDE error codes and also the native error codes of the SQL server you are connected to. Besides the Message property, the EDBEngineError class has two more properties, ErrorCount and Errors. This last property is a list of errors:

```
property Errors[Index: Integer]: TDBError;
```

Each item within this list is an object of the class TDBError, which has the following properties:

```
type
  TDBError = class
    ...
  public
    property Category: Byte read GetCategory;
    property ErrorCode: DBIResult read FErrorCode;
    property SubCode: Byte read GetSubCode;
    property Message: string read FMessage;
    property NativeError: Longint read FNativeError;
  end;
```

I've used this information to build a simple database program showing the details of the errors in a memo component. To handle all of the errors, the DBError example installs a handler for the OnException event of an ApplicationEvents component. The event handler simply calls a specific method used to show the details of the database error, in case it is an EDBEngineError:

```
procedure TForm1.ApplicationEvents1Exception (Sender: TObject; E: Exception);
begin
  Beep;
  if E is EDBEngineError then
    ShowError (EDBEngineError (E))
  else
    ShowMessage (E.Message);
end;
```

I decided to separate the code used to show the error to make it easier for you to copy this code and use it in different contexts. Here is the code of the ShowError method, which outputs all of the available information to the Memo1 component that I've added to the form:

```
procedure TForm1.ShowError(E: EDBEngineError);
var
  I: Integer;
begin
```

```
    Memo1.Lines.Add('');
    Memo1.Lines.Add('Error: ' + (E.Message));
    Memo1.Lines.Add('Number of errors: ' + IntToStr(E.ErrorCount));
    // iterate through the Errors records
    for I := 0 to E.ErrorCount - 1 do
    begin
      Memo1.Lines.Add('Message: ' + E.Errors[I].Message);
      Memo1.Lines.Add('   Category: ' + IntToStr(E.Errors[I].Category));
      Memo1.Lines.Add('   Error Code: ' + IntToStr(E.Errors[I].ErrorCode));
      Memo1.Lines.Add('   SubCode: ' + IntToStr(E.Errors[I].SubCode));
      Memo1.Lines.Add('   Native Error: ' + IntToStr(E.Errors[I].NativeError));
      Memo1.Lines.Add('');
    end;
  end;
```

Besides this error-handling code, the program has a table and a query, along with the error-related event handlers. As already mentioned, you can install an event handler related to specific errors of a dataset. The three events OnPostError, OnDeleteError, and OnEditError have the same structure. Their handlers receive as parameters the dataset, the error itself, and an action you can request from the system; this can be set to daFail, daAbort, or daRetry:

```
procedure TForm1.Table1PostError(DataSet: TDataSet; E: EDatabaseError;
  var Action: TDataAction);
begin
  Memo1.Lines.Add (' -> Post Error: ' + E.Message);
end;
```

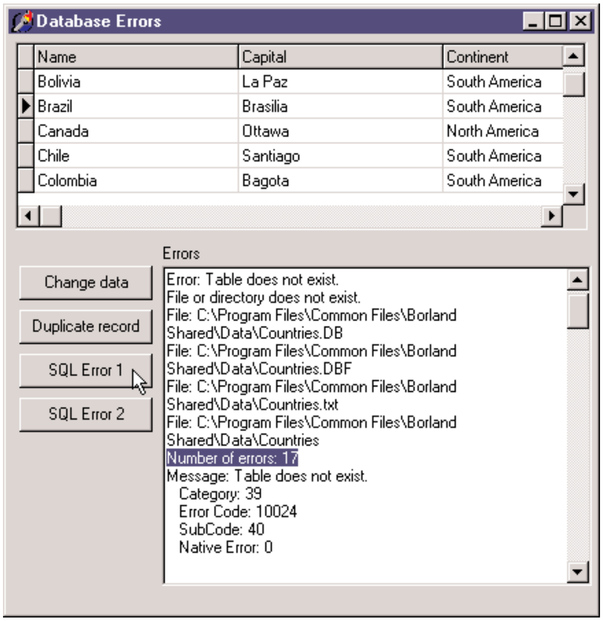
If you don't specify an action, as in the code above, the default daFail is used, and the exception reaches the global handler. Using daAbort stops the exception and can be used if your event handler already displays a message. Finally, if you have a way to determine the cause of the error and fix it, you can use the daRetry action.

NOTE The fourth error event, OnUpdateError, has a different structure and is used along with cached updates as the information is sent back from the local cache to the database. This handler is important for handling update conflicts among different users as described in the next example.

The example has also a DBGrid connected with the table. You can use the DBGrid to perform some illegal operations, such as adding a new record with the same key as an existing one or trying to execute illegal SQL queries. Pressing the four buttons on the left of the memo generate errors, as you can see in Figure 13.23.

FIGURE 13.23:

The third button of the DBError form generates an exception with 17 database errors!



Name	Capital	Continent
Bolivia	La Paz	South America
Brazil	Brasilia	South America
Canada	Ottawa	North America
Chile	Santiago	South America
Colombia	Bagota	South America

```
Error: Table does not exist.
File or directory does not exist.
File: C:\Program Files\Common Files\Borland
Shared\Data\Countries.DB
File: C:\Program Files\Common Files\Borland
Shared\Data\Countries.DBF
File: C:\Program Files\Common Files\Borland
Shared\Data\Countries.txt
File: C:\Program Files\Common Files\Borland
Shared\Data\Countries
Number of errors: 17
Message: Table does not exist.
   Category: 39
   Error Code: 10024
   SubCode: 40
   Native Error: 0
```

What's Next?

In this chapter, we have seen examples of database access from Delphi programs. I have covered the basic data-aware components as well as the development of database applications based on standard controls. We've explored the internal architecture of the TDataSet class and of field objects, and discussed many of the events and properties shared by all datasets and used by all database applications. Even though most of the demonstrations used BDE tables and queries, in the entire text there was little or no code specific to those components. Most of what I've described equally applies to client/server applications, built with BDE, dbExpress, IBX, ADO or other dataset components.

We've discussed calculated fields, lookup fields, customizations of the DBGrid control, and many rather advanced techniques. What we really haven't delved into is the database and data-access side of the picture, which depends on the actual type of database engine and server you plan using. The next chapter will start focusing in this topic, with an in-depth overview of client/server development. Following chapters will put the accent on specific data-access technologies and provide even more advanced information.

CHAPTER 14

Client/Server Programming

- Overview of client/server
- Porting local applications
- Elements of database design
- Client/server with BDE
- The dbExpress library
- Using the ClientDataSet component
- Local databases with MyBase

In the last chapter, we examined Delphi's support for database programming, using local files (particularly Paradox) in most of examples but not focusing on any specific database technology. This chapter moves on to the use of SQL server databases, focusing on client/server development with the BDE and the new dbExpress technology. A single chapter cannot cover this complex topic in detail, so I'll simply introduce it from the perspective of the Delphi developer and add some tips and hints. The next chapter extends our discussion of client/server programming, providing some real-world examples. I'll use InterBase in both chapters, because this is the RDBMS (relational database management system), or SQL server, that is included in the Enterprise edition of Delphi and because it is a free and open-source server.

In a rapid application development tool such as Delphi, you can indeed take the same components and code developed for a local database application and use them in a client/server environment. However, this handy feature may prove to be dangerous to beginners, as a standard technique that works well for local access might become extremely inefficient in a client/server application.

An Overview of Client/Server Programming

The database applications in previous chapters used the BDE to access data stored in files either on the local machine or on a networked computer. In both cases we used a file server, whose only role was to store the file on a hard disk, because the database engine (the BDE) was running exclusively on the computer that also hosted the application. In this configuration, when we query one of the tables, its data is first copied into a local cache of the BDE and then processed.

As an example, consider taking a table like EMPLOYEE (part of the InterBase sample database, which ships with Delphi), adding thousands of records to it, and placing it on a networked computer working as a file server. If we want to know the highest salary paid by the company, we can open a Table component (`EmpTable`) connected with the database table and run this code:

```
EmpTable.Open;
EmpTable.First;
MaxSalary := 0;
while not EmpTable.Eof do
begin
  if EmpTable.FieldByName ('Salary').AsCurrency > MaxSalary then
    MaxSalary := EmpTable.FieldByName ('Salary').AsCurrency;
  EmpTable.Next;
end;
```

The effect of this approach is to move all the data of the (large) table from the networked computer to the local machine, an operation that might take minutes. Because Delphi includes a Query component, you might think of using the following SQL code to compute this maximum value:

```
select Max(Salary) from Employee
```

In case of a local table, this query would be processed by the local SQL engine of the BDE, and the entire dataset of the table would still have to be moved from the networked computer to the local one, with similarly poor performance. But if you use InterBase and let the server execute the SQL code, only the result set—a single number—will need to be transferred to the local computer.

NOTE The two code excerpts above are part of the GetMax example, which includes some code to time the two approaches. Using the Table component on the small EMPLOYEE table takes about ten times longer than using the query, even if the InterBase server is installed on the client computer.

If you want to store a large amount of data on a central computer and avoid moving the data to client computers for processing, the only solution is to let the central computer manipulate the data and send back to the client only a limited amount of information. This is the foundation of client/server programming.

In general, you'll use an existing program on the server (an RDBMS) and write a custom client application that connects to it. Sometimes, however, you might even want to write both a custom client and a custom server, as in three-tier applications. Delphi support for this type of program—what has been called the MIDAS architecture—is covered in Chapter 17, "Multitier Database Applications with DataSnap."

The *upsizing* of an application—that is, the transfer of data from local files to a SQL server database engine—is generally done for performance reasons and to allow for larger amounts of data. Going back to the previous example, in a client/server environment, the query used to select the maximum salary would be computed by the RDBMS, which would send back to the client computer only the final result, a single number! With a powerful server computer (such as a multiprocessor Sun SparcStation), the total time required to compute the result might be minimal.

However, there are also other reasons to choose a client/server architecture:

The amount of data A Paradox table cannot exceed 2 GB, but even around 300 MB you might start having serious speed problems, and errors in the indexes become more frequent.

The need for concurrent access to the data Paradox uses the `Paradox.NET` file to keep track of which user is accessing the various tables and records. The Paradox approach to handling multiple users is based on *pessimistic locking*. When a user starts an editing operation on a record, none of the other users can do the same (to avoid any update conflict), as we saw in the last chapter. In a system with tens of users, this might lead to serious problems, because a single user might block the work of many others. SQL server databases, by contrast, generally use *optimistic locking*, an approach that allows multiple users to work on the same data and delays the concurrency control until the time the users send back some updates.

Protection and security An RDBMS usually has many more protection mechanisms than the simple password you can add to a Paradox table. When your application is based on files, a malicious or careless user might simply delete those vital files. When SQL servers are based on robust operating systems, instead, they provide multiple levels of protection, make backup easier, and often allow only the database administrator to modify the structure of the tables.

Programmability An RDBMS database can host business rules, in the form of stored procedures, triggers, table views, and other techniques we'll discuss in this and the next chapter. Choosing how to divide the application code between the client and the server is one of the main issues of client/server programming.

Transaction control Local files offer some support for transactions, but the transaction support provided by an RDBMS database is generally much greater. This is another important aspect of the overall robustness of the system.

From Local to Client/Server

Now we can start focusing on particular techniques useful for client/server programming. Keep in mind that the general goal is to distribute the workload properly between the client and the server and reduce the network bandwidth required to move information back and forth.

The foundation of this approach is good database design, which involves both table structure and appropriate data validation and constraints, or business rules. Enforcing the validation of the data on the server is important, as the integrity of the database is one of the key aims of any program. However, the client side should include data validation as well, to improve the user interface and make the input and the processing of the data more user-friendly. It makes little sense to let the user enter invalid data and then receive an error message from the server, when we can prevent the wrong input in the first place.

NOTE If you use a CASE tool for the definition of the database, or import the definition in such a tool afterward, you can use Delphi's Case Wizard to generate a corresponding data dictionary and have the field objects created at design time automatically import the constraints specified on the server.

Unidirectional Cursors

In local databases, tables are sequential files whose order is either the physical order or is defined by an index. By contrast, SQL servers work on logical sets of data, not related to a physical order. A *relational* database server handles data according to the relational model, a mathematical model based on set theory.

What is important for the present discussion is that in a relational database, the records (sometimes called tuples) of a table are identified not by position but exclusively through a primary key, based on one or more fields. Once you've obtained a set of records, the server adds to each of them a reference to the following one, which makes it fast to move from a record to the following one but terribly slow to move back to the previous record. For this reason, it is common to say that an RDBMS uses a *unidirectional* cursor. Connecting such a table or query to a DBGrid control is practically impossible, as this would make it terribly slow when browsing the grid backward.

The BDE helps a lot to handle unidirectional cursors, as it keeps in a local cache the records already loaded in the table. Thus, when we move to following records, they are requested from the SQL server; but when we go back, the BDE jumps in and provides the data. In other words, the BDE makes these cursors fully bidirectional, although this might use quite a lot of memory. When using dbExpress, which doesn't provide a similar caching system, a program needs to keep in memory the records it has already accessed. This can be easily accomplished by means of the ClientDataSet component.

NOTE The simple case of a DBGrid used to browse an entire table is common in local programs but should generally be avoided in a client/server environment. It's better to filter out only part of the records and only the fields you are interested in. Do you need to see a list of names? Return all those starting with the letter *A*, then those with *B*, and so on, or ask the user for the initial letter of the name.

If proceeding backward might result in problems, keep in mind that jumping to the last record of a table is even worse; usually this operation implies fetching all the records! A similar situation applies to the `RecordCount` property of datasets. Computing the number of records often implies moving them all to the client computer. This is the reason why the thumb of the vertical scrollbar of the DBGrid works for a local table but not for a remote one. If you need to know the number of records, run a separate query to let the server (and not the client) compute it. For example, you can see how many records will be selected from the EMPLOYEE table if you are interested in those records having a salary field higher than 50,000:

```
select count(*)
from Employee
where Salary > 50000
```

TIP Using the SQL instruction count(*) is a handy way to compute the number of records returned by a query. Instead of the * wildcard, we could have used the name of a specific field, as in count(First_Name), possibly combined with either distinct or all, to count only records with different values for the field or all the records having a non-null value.

Parametric Queries and Null Values

Parametric queries are a very useful technique. Essentially, they allow you to run multiple queries with different result sets while the server only needs to work out the access strategy for solving the query once.

You can force this initial preparation of the query access strategy by calling the Prepare method of a Query component. With this operation, the server receives the query, checks its syntax, and while compiling it determines how to use indexes and other access techniques. Multiple executions of the query will be faster because these initial operations have already been executed once for all. Of course, you should call Prepare again if you change the SQL text of the query. Also, remember to call Unprepare at the end, to free some BDE resources.

Note that some powerful SQL servers can do the same operation by caching the requests and automatically determining that you are sending the same request twice. If the server is smart enough, preparing the query might result in little or no performance gain.

NOTE When you write parametric queries against a SQL server, you should consider null values with care. In fact, to test for a null value, you should not write a field = null test, but use the specific expression field is null instead.

Elements of Database Design

Although this is a book on Delphi programming and not on databases, I feel it's quite important to discuss a few elements of good (and modern) database design. The reason is simple: if your database design is incorrect or convoluted, you'll either have to write terribly complex SQL statements and server-side code, or write a lot of Delphi code to access your data, possibly even fighting against the design of the TDataSet class.

Entities and Relations

The *classic* relational database design approach, based on the entity–relation (E-R) model, involves having one table for every entity you need to represent in your database, with one field for each data element you need plus one field for every one-to-one or one-to-many relation to another entity (or table). For many-to-many relations, instead, you'll need a separate table.

As an example of a one-to-one relation, consider a table representing a university course. This would have a field for each relevant data element (name and description, room where it is held, and so on) plus a single field indicating the teacher. The data of the teacher, in fact, should not be stored within the course data, but in a separate table, as it might be referenced from elsewhere.

The schedule of each course can include an undefined number of hours in different days, so they cannot be added inside the same table describing the course. Instead, this information must be placed on a separate table, including all of the schedules, with a field referring to the class each schedule is for. In a one-to-many relation, such as this, "many" records of the schedule table point back to the same "one" record of the course table.

A more complex situation is required to store which student is taking which class. Students cannot be listed directly in the course table, as their number is not fixed, and the classes cannot be stored within the student's data for the same reason. In a similar many-to-many relation, the only approach is to have an extra table representing the relation, which lists references to students and courses.

Normalization Rules

The *classic* design principles include a series of so-called *normalization* rules. The goal of these rules is to avoid duplicating data in your database (not only for saving space, but mainly to avoid ending up with incongruous data). For example, you don't repeat all of the customer details in each order, but refer to a separate customer entity. This way you save memory, but when the customer details change (for example, because of a change of address) all of the orders of this customer will reflect the new data. Other tables that relate to the same customer will probably be automatically updated as well.

Normalization rules imply using codes for commonly repeated values. For example, if you have a few different shipment options, you won't use a string-based description for these options within the orders table, but would rather use a short numeric code, mapped to a description in a separate lookup table.

This last rule, which should not be taken to the extreme, is to avoid having to join a large number or table for every query. You can either account for some de-normalization (leaving a short shipment description within the orders table) or use the client program to provide the description, again ending up with a formally incorrect database design. This last option is practical only when you use a single development environment (let's say, Delphi) to access this database.

From Primary Keys to OIDs

In a relational database, records are not identified by a physical position (as in Paradox and other local databases) but only by the data within the record itself. Typically, you don't need the data of all of the fields to identify a record, but only a subset of the data, forming the so-called *primary key*. If the fields that are part of the primary key must identify an individual record, their value must be different for each possible record of the table.

NOTE Technically, many database servers add internal record identifiers to the tables, but this happens only for internal optimizations and has little to do with the logical design of a relational database. Also, these internal identifiers work differently in different SQL servers and might even change among versions, a good reason not to rely on them.

The early incarnations of the relational theory dictated the use of *logical keys*, which means selecting one or more records that indicate an entity without risk of any confusion. This is often easier to say than to accomplish. For example, company names are not generally unique, and even the company name and its location don't provide a complete guarantee. Moreover, if a company changes its name (not an unlikely event, as Borland can teach us) or its location, and you have references to the company within other tables, you must change all those references as well, with the risk of ending up with dangling references.

For this reason, and also for efficiency (using strings for references implies using a lot of space in secondary tables, where references often occurs), logical keys have been invariably phased out for physical or surrogate keys. *Physical keys* refer to a single field of the table identifying an element in a unique way. For example, each person in the U.S. has a Social Security number (SSN), but almost every country has a tax ID or other government-assigned number that identifies each person. The same typically exists for companies. Although these ID numbers are guaranteed to be unique, they might change depending on the country (creating troubles for the database of a company also selling its goods abroad) or even within a single country (to account for new tax laws). They are also often inefficient, as they might be quite large (Italy, for example, uses a 16-character code, letters and numbers, to identify people).

Another common approach is to use *surrogate keys*, which are basically numbers identifying each record, in the form of client codes, order numbers, and so on. These surrogate keys are commonly used in database design. However, in many cases, these end up being some sort of *logical identifiers*, with client codes showing up all over the places, not a great idea overall.

WARNING The situation becomes particularly troublesome when these surrogate keys also have a meaning and must follow specific rules. For example, companies must number invoices with unique and consecutive numbers, without leaving holes in the numbering sequence. This situation is extremely complex to handle programmatically, if you consider that only the database can determine these unique consecutive numbers when we send new data to it. At the same time, we need to identify the record before we send it to the database, otherwise we won't be able to fetch it again. Practical examples of how to solve this situation are discussed in the next chapter.

OIDs to the Extreme

An extension to the use of surrogate keys is the use of a unique identifier, also called object identifier (OID). An OID is a number, or a string with sequence of numbers and digits, added to each record of each table representing an entity (and at times even to records of tables representing relations). Differently from client codes, invoice numbers, SSN, or purchase order numbers, OIDs are totally random, without any sequencing rule, and never visible to the end user. This means you can still use surrogate keys (if your company is used to them) along with OIDs, but all the external references to the table will be based on OIDs.

Another common rules suggested by the promoters of this approach (which is part of the theories supporting object-relational mapping) is the use of system-wide unique identifiers. If you have a table of client companies and a table of employees, you might wonder why you should use a unique ID for such diverse data. The reason is that, if you do so, you'll be able to sell goods to an employee without having to duplicate the employee information into the customers table, but simply referring to the employee in your order and invoice. An order is placed by someone identified by an OID, and this OID can refer to many different tables (but of course not all of them).

NOTE Using OIDs and the object-relational mapping is an advanced element of the design of Delphi database applications. My personal suggestion is to investigate this topic before embracing medium or large-size Delphi projects, as the benefit can be relevant (after some investment in studying this approach and building some basic support code).

External Keys and Referential Integrity

Getting back to the standard database design, the keys identifying a record (whichever their type) can be used as external keys in other tables, for example to represent the various types of relations discussed earlier. All SQL servers are capable of verifying these external references, so that you cannot refer to a nonexistent record of another table. These referential integrity constraints are expressed when you create a table.

Besides not being allowed to add references to nonexistent records, you're generally prevented from deleting a record if there are external references to it. Some SQL servers go one step further: As you delete a record, instead of simply denying the operation, they can automatically delete all records that refer to it from other tables.

More Constraints

Besides the uniqueness of primary keys and the referential constraints, you can generally use the database to impose more validity rules on the data. You can ask for specific columns (such as those referring to a tax ID or a purchase order number) to include only unique values. You can impose uniqueness of the values of multiple columns—for example, to indicate you cannot run two classes in the same room at the same time.

In general, simple rules can be expressed imposing constraints on a table, while more complex rules generally imply the execution of stored procedures activated by triggers (every time the data changes, for instance, or there is new data).

Again, there is much more to proper database design, but the simple elements discussed in this section can provide a starting point, or a good refresher.

NOTE For more on the Data Definition Language and Data Manipulation Language of SQL, see the bonus chapter "Essential SQL" on the *Mastering Delphi* CD.

Client/Server with the BDE

Now let's consider how Delphi fits into the client/server picture. How does it help us build client/server applications? As I've mentioned, you can still use all the components and techniques discussed in the Chapter 13, "Delphi's Database Architecture," although in some cases alternate approaches will help you leverage the power of the RDBMS your application is dealing with.

As a starting point, let's cover a few considerations on Delphi client/server development using the BDE and its components. After this I'll move to dbExpress, which in Delphi 6 is the recommended general solution for client/server development.

NOTE For a list of the alternative approaches Delphi 6 offers for database access, see the initial part of the preceding chapter.

SQL Links

The BDE doesn't know how to handle the RDBMS; it uses some further drivers, called SQL Links, to perform this operation. As an alternative, the BDE can also interact with ODBC drivers. Borland provides native BDE drivers for InterBase, Oracle, Informix, Microsoft SQL Server, Sybase, and DB2.

If the BDE is still required on the local machines, it can actually be very efficient. For example, when you use the pass-through mode for queries, the BDE doesn't try to interpret the SQL code but passes it directly to the RDBMS server. This allows you to use a server's specific SQL commands and also to speed up the execution. The pass-through mode is activated using the BDE Administrator utility.

Having the BDE between the client and the server can also help in building applications designed to work with multiple servers. In practice, however, it's not easy to do this and still obtain the best performance, because of differences in the SQL dialects understood by each SQL server. In particular, data types are handled differently by the various servers. If the same table were placed on two servers that have data type differences, Delphi would need to use two different TField objects (which creates a few headaches if you want to define the fields at design time).

The Database Component

In local BDE applications, programmers usually refer to the database by indicating the alias of the file path in the DatabaseName property of the Table and Query components. A better approach is to use the Database component to define a local alias and then let all the DataSet components refer to this local alias.

As an example, consider the components of the GetMax application mentioned at the beginning of this chapter:

```
object Database1: TDatabase
  AliasName = 'IBLOCAL'
  Connected = True
  DatabaseName = 'IB'
  LoginPrompt = False
  Params.Strings = (
    'USER NAME=SYSDBA'
    'PASSWORD=masterkey')
  SessionName = 'Default'
end
object EmpTable: TTable
  DatabaseName = 'IB'
  TableName = 'EMPLOYEE'
end
```

```
object EmpQuery: TQuery
  DatabaseName = 'IB'
  SQL.Strings = (
    'select Max(Salary) from Employee ')
end
```

In a client/server application, using the Database component is almost mandatory, as it is required to define connectivity and login parameters (the user name and password, as you can see in the `Params` property above) and to handle transactions.

Keep in mind that the Database component establishes a connection with the RDBMS, representing one of the clients of the system. As such, on most servers it requires a license, and your organization is typically paying for a fixed number of licenses. If the same application or the same computer uses multiple connections to the server, it can count as multiple clients! Fortunately, by setting the `KeepConnection` property of the Database component, you can specify whether to keep the database connection active even when there is no active DataSet component using the connection. If your program can fetch some data and then operate on it locally, disconnecting from the server might help you conserve licenses.

BDE Table and Query Components in Client/Server

In Delphi there are two BDE components you use to access an existing database table: Table and Query. When building client/server applications, programmers tend to use the Query component exclusively, but that is certainly not mandatory, and there are cases in which using the simpler Table component has no drawback. Here's a quick look at the pros and cons of both components:

- While the Table component should not be used to access a large table, it can work perfectly well with a small lookup table. By opening a Table component, you don't transfer the entire content of the table to the local machine; the data is moved only when you access specific records.
- Consider also that with the Table component, the BDE asks the server first for the table structure and then for the table data. These two steps are necessary for setting up the proper internal structures of the BDE, and they are not executed by the Query component. If you activate the BDE's Schema Caching feature, the logical structure of the table will be kept locally, saving this extra step. Of course, this might create problems if the logical structure of the table changes on the server.
- One problem with the Table component is that the BDE mimics a bidirectional cursor by caching the data locally. With a Query component, instead, you can specify whether you want this caching or not with the `Unidirectional` property.

- Another point to consider is that you can generally edit the result of a simple query, sending the data back to the SQL server. This is accomplished by setting the `RequestLive` property to True. For more complex queries, however, you'll need to use an UpdateSQL component, something we'll discuss later in this chapter.
- When trying to minimize the data moved between the server and the client, you need to consider the size of each record as well as the total number of them. When you select only a few fields with a query, only part of the data is considered. A Table component, instead, always entails transferring the entire record to the local machine, even if you've filtered out some fields using the Fields editor. The same problem takes place when you ask for a live query (by setting the `RequestLive` property). In this case, the BDE needs to see the entire record in order to send back the proper update commands. This means that selecting all the records of a table with a live query is equivalent to using the Table component.
- The Query component is not limited to `select` SQL statements; you can also use it to insert or delete records. When the Query component returns a dataset, you generally activate it with the `Open` method (or with the equivalent operation, setting the `Active` property to True). When the Query component is used to perform an operation on the server, you activate it by calling the `ExecSQL` method.

Using Table and Query Filters

One way to limit the amount of data returned by a table is to filter it. Using the `Filter` property of the Table component, you can specify a condition similar to the `where` clause of a query. When you work with local databases, the filter is applied by the BDE, but with a SQL server, the BDE passes the condition to the server in the query generated for the table. This makes filtered tables very portable between local and client/server applications.

WARNING The situation is different if you filter the records in the Pascal code, using the `OnFilterRecord` event. In this case, all the records are sent to the client application, which does its own custom filtering.

If you use a filter with a Query component, the filtering operation will always be performed locally by the BDE, even when you are working with a SQL server. In this case, the BDE asks the server for the entire result set of the query. This would be reasonable only when the user of the application changes the filtering condition often. For a query, only the local filter will be modified, and the data in the local cache will be used. For a table, the BDE will generate an updated query to be executed.

Live Queries and Cached Updates

When working with local data, it is very common to use grids and other visual controls, edit the data, and send it back to the database. We've already seen that using a DBGrid might cause problems when working with an RDBMS, as moving on the grid might send numerous data requests to the server, creating a huge amount of network traffic.

When you use the Query component to connect to some data, you cannot edit the data unless its `RequestLive` property is set to True. If you are working with local tables, the query is always elaborated by the BDE with the Local SQL engine. The BDE will allow for a live query only if it is quite simple: All joins should be outer joins; there cannot be a `distinct` key; there can be no aggregation, no `group by` or `having` clause, no subqueries, and no `order by` unless supported by an index; and there are other rules you can find in Delphi's Help.

If you are working with a SQL server, setting a live query will put the BDE in control of the query, instead of the server. When connected to a SQL server, a live query behaves like a Table component. (So it makes sense to use the table anyway, in these cases.)

TIP Most SQL servers, including InterBase, allow you to define updateable views based on the result of a `select` statement that the Local SQL engine of the BDE won't consider updateable. Then you can simply hook a Table component to the view, letting the SQL server do the work and bypassing the Local SQL engine of the BDE.

If the BDE determines that the dataset cannot be updated, it sets the `CanModify` property to False. The DataSource component checks this value before allowing an editing operation. A solution to this problem is to avoid the use of data-aware controls, as discussed in the last chapter, and use specific SQL queries to update, insert, and delete records.

A better approach is to automate this process (retaining the capabilities of the data-aware controls) by using the UpdateSQL component together with the Query component. The UpdateSQL can be used only in conjunction with cached updates, a topic discussed in the last chapter. The basic idea is that the update operations are kept in a local cache until the program calls the `ApplyUpdates` method of the Query component. This operation corresponds to the execution of a series of `update`, `insert`, and `delete` SQL operations on the server, using the data in the cache. The required SQL commands are held by the UpdateSQL component, which has a design-time editor you can use to generate these SQL commands almost automatically.

Cached updates solve the live queries issue, reduce network traffic, define a standard way to solve updates conflicts, and reduce the server load, but they require more memory on the client computer.

TIP A much better approach than using cached updates is to rely on the ClientDataSet component, which is extremely powerful and allows you to do similar things yet leaving you a lot more programmatic control.

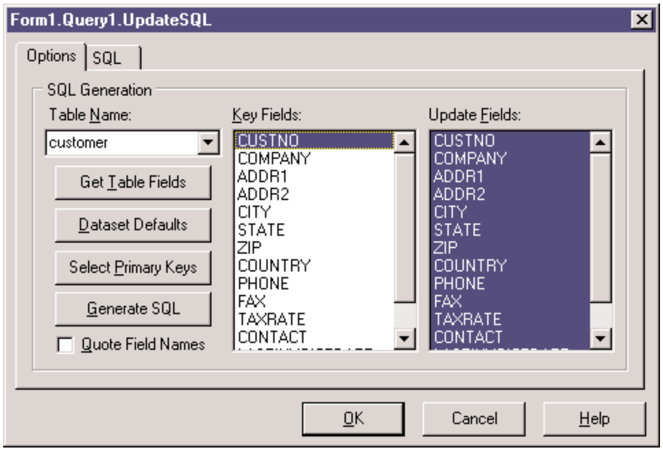
The UpdateSQL Component

The role of the UpdateSQL component is to provide a query with the update statements required to make its result set editable. Its key properties are `DeleteSQL`, `InsertSQL`, and `ModifySQL`, but the most important element is the `UpdateObject` property of the related Query component. The update SQL statements are executed when you apply the cached updates, sending the changes to the server. Because cached updates maintain the information on the original records, the updates usually indicate which record to update by passing the original data. This is the only way we have to identify a record on a SQL server, and this technique also helps the server to track any updates on the same record done by other users.

All this setup might seem to imply a lot of work, but it is actually very simple. After you've written a query, you can connect the UpdateSQL component to it and activate the component editor, as shown in Figure 14.1.

FIGURE 14.1:

The UpdateSQL component editor in action



This component editor has two tabs. The first indicates the criteria used to generate the SQL statements for adding, deleting, or modifying records. With a join, you can select the table to update and the fields involved. When you've completed this step, click the Generate SQL button and the editor will move to the second tab, where you can inspect the generated SQL code for the three operations.

The UpdateSQL Example

To demonstrate the real power of the UpdateSQL component, I've built a complex example called UpdateSQL, based on the Employee, Department, and Job tables of the IBLocal database we've used in the past. Here is the textual description of the UpdateSQL component of the example:

```
object EmpUpdate: TUpdateSQL
  ModifySQL.Strings = (
    'update EMPLOYEE'
    'set'
    '   FIRST_NAME = :FIRST_NAME,'
    '   LAST_NAME = :LAST_NAME,'
    '   SALARY = :SALARY,'
    '   DEPT_NO = :DEPT_NO,'
    '   JOB_CODE = :JOB_CODE,'
    '   JOB_GRADE = :JOB_GRADE,'
    '   JOB_COUNTRY = :JOB_COUNTRY'
    'where'
    '   EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '   (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE,
    '   JOB_GRADE, JOB_COUNTRY)'
    'values'
    '   (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, '
    '   :JOB_GRADE, :JOB_COUNTRY)')
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where'
    '   EMP_NO = :OLD_EMP_NO')
  end
```

To delete the employee records, the program uses a stored procedure, which is already available in the sample database and is connected to the following component:

```
object spDelEmployee: TStoredProc
  DatabaseName = 'AppDB'
  StoredProcName = 'DELETE_EMPLOYEE'
  ParamData = <
    item
      DataType = ftInteger
      Name = 'EMP_NUM'
      ParamType = ptInput
    end>
end
```

The OnUpdateRecord event of the Query component uses the stored procedure instead of the default UpdateSQL component for deleting records. Here is the code of the event handler:

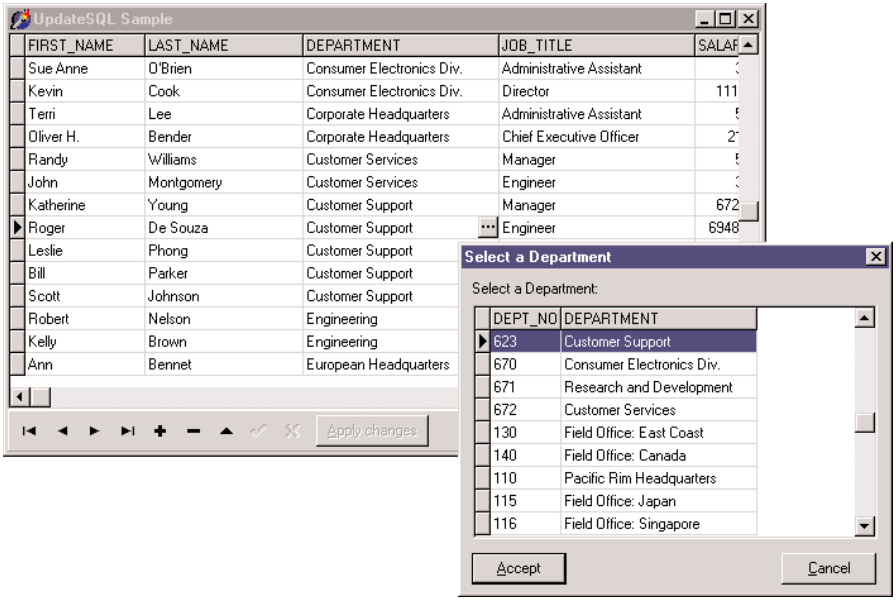
```
procedure TdmData.qryEmployeeUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  // when deleting the record, use the stored procedure
  if UpdateKind = ukDelete then
  begin
    // assign emp_no value
    with dmData do
      spDelEmployee.Params[0].Value := qryEmployeeEMP_NO.OldValue;
    try
      // invoke stored procedure that tries to delete employee
      dmData.spDelEmployee.ExecProc;
      UpdateAction := uaApplied; // success
    except
      UpdateAction := uaFail;
    end;
  end
  else
  try
    // apply updates
    dmData.EmpUpdate.Apply(UpdateKind);
    UpdateAction := uaApplied;
  except
    UpdateAction := uaFail;
  end;
end;
```

Notice that because we perform the update operation directly, we must indicate in the UpdateAction parameter whether it succeeds or not. This code is part of the data module. The main form, visible at run time in Figure 14.2, has a couple of extra features. If the user closes the form with any updates pending, the OnCloseQuery event of the form displays a warning message, allowing the user to apply the updates or skip them:

```
procedure TMainForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
var
  Res: Integer;
begin
  with dmData do
    if qryEmployee.UpdatesPending then
    begin
      Res := MessageDlg (CloseMsg, mtInformation, mbYesNoCancel, 0);
      if Res = mrYes then
        AppDB.ApplyUpdates ([qryEmployee]);
      CanClose := Res <> mrCancel;
    end;
end;
```

FIGURE 14.2:

The main form of the UpdateSql example along with a secondary form



FIRST_NAME	LAST_NAME	DEPARTMENT	JOB_TITLE	SALAF
Sue Anne	O'Brien	Consumer Electronics Div.	Administrative Assistant	3
Kevin	Cook	Consumer Electronics Div.	Director	111
Terri	Lee	Corporate Headquarters	Administrative Assistant	5
Oliver H.	Bender	Corporate Headquarters	Chief Executive Officer	27
Randy	Williams	Customer Services	Manager	5
John	Montgomery	Customer Services	Engineer	3
Katherine	Young	Customer Support	Manager	672
Roger	De Souza	Customer Support	Engineer	6948
Leslie	Phong	Customer Support		
Bill	Parker	Customer Support		
Scott	Johnson	Customer Support		
Robert	Nelson	Engineering		
Kelly	Brown	Engineering		
Ann	Bennet	European Headquarters		

Select a Department:

DEPT_NO	DEPARTMENT
623	Customer Support
670	Consumer Electronics Div.
671	Research and Development
672	Customer Services
130	Field Office: East Coast
140	Field Office: Canada
110	Pacific Rim Headquarters
115	Field Office: Japan
116	Field Office: Singapore

The second feature is the use of a secondary form to update the fields that are related to other tables—the fields involved in the joins. The program uses two secondary dialog boxes, which get the data from other two Query components. The dialog boxes are displayed when the user clicks the ellipsis button of the DBGrid control, in the `OnEditButtonClick` event. Here is the first part of this event handler, related to the selection of the department:

```
procedure TMainForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  // check whether this is the department field
  if DBGrid1.SelectedField = dmData.qryEmployeeDEPARTMENT then
    with TfrmDepartments.Create(self) do
    try
      dmData.qryDepartment.Locate('DEPT_NO',
        dmData.qryEmployeeDEPT_NO.Value, []);
      if ShowModal = mrOk then
        with dmData do
        begin
          if not (qryEmployee.State in [dsEdit, dsInsert]) then
            qryEmployee.Edit;
          qryEmployeeDEPT_NO.Value := qryDepartment.Fields[0].Value;
          qryEmployeeDEPARTMENT.Value := qryDepartment.Fields[1].Value;
        end;
    finally
      Free;
    end
  else // similar code for the job fields...
```

Finally, the Apply button simply calls the ApplyUpdates method if there are pending updates and then refreshes the data of the query:

```
procedure TMainForm.btnApplyClick(Sender: TObject);
begin
  with dmData do
    if qryEmployee.UpdatesPending then
    begin
      AppDB.ApplyUpdates([qryEmployee]);
      // refresh the data
      qryEmployee.Close;
      qryEmployee.Open;
      btnApply.Enabled := False;
    end;
end;
```

If you run this program, you'll notice that even if the underlying query is read-only, you can change data directly in the DBGrid, as you would do with a regular Table component. The visual operations you do are temporarily stored in the cache; then, when you issue the update operation, the UpdateSQL and the StoredProc components provide the actual code. Also keep in mind that the Salary field has some constraints (defined in the sample database), so you have to change it carefully to avoid errors on the server when the changes are applied.

Update Conflicts

When you are working with local tables, using cached updates might cause concurrency problems. A plain edit operation usually places a lock on the table, so that the other users cannot modify the same record until the first user has posted the updates. The previous chapter covered locking and concurrency issues in detail.

When working with SQL servers, however, the default locking behavior is optimistic. Multiple users can update the same records, and only when the data is sent back does the server verify the original data of the record before updating it, potentially raising an error. More precisely, the update statement uses one or more original fields to locate the record you want to update. If you use all fields and another user has changed the record, then the server will not find the original record and will cause an update error.

You can manually control this behavior either in the code of the UpdateSQL component (indicating to include all the fields read in the query) or by using the UpdateMode property of the Table and Query components. The default value, upWhereAll, indicates that the update query will have a where clause with all the original fields of the record. In many cases, the fact that another user has modified a field different from those we have modified is not an error. We can set the upWhereChanged mode to let Delphi generate an exception and show an error message only if the current and the other user have both modified the same fields. The third alternative is to use the key field only to identify the record, which means that update conflicts will be ignored and that the last user posting the data will simply override any previous change. As you can imagine, this is generally an option to avoid in a client/server, multiuser environment.

Using Transactions

Whether you are working with a SQL server, you should use *transactions* to make your applications more robust. The idea of a transaction can be described as a series of operations to be considered as a single, "atomic" whole that cannot be split.

An example may help to clarify the concept. Suppose you have to raise the salary of each employee of a company by a fixed rate, as we did in the Total example of the preceding chapter. Now if during the operation an error occurs, you might want to undo the previous changes. If you consider the operation "raise the salary of each employee" as a single transaction, it should either be completely done or completely ignored. Or consider the analogy with financial transactions—if only part of the operation is performed, because of an error, you might end up with a missed credit or with some extra money!

Working with database operations as transactions serves a useful purpose. You can start a transaction and do several operations that should all be considered parts of a single larger operation; then, at the end, you can either commit the changes or *roll back* the transaction, discarding all the operations done up to now. Typically, you might want to roll back a transaction if an error occurred during its operations.

Handling transactions in Delphi is quite simple. By default, each edit/post operation is considered a single *implicit* transaction, but you can alter this behavior by handling them explicitly. Simply use the following three methods of the BDE Database component (other database connection components have similar methods):

StartTransaction marks the beginning of a transaction.

Commit confirms all the updates to the database done during the transaction.

Rollback returns the database to its state prior to starting the transaction.

The Database component determines the transaction isolation level using the `TransIsolation` property. When one user starts a transaction and modifies data, should such changes be visible to other users? And what happens if the user rolls back the transaction? To such questions there isn't a universal answer; every programmer should try to answer them according to the requirements or business rules of the application. There are three alternative values for transaction isolation in the BDE:

tiDirtyRead makes the updates of a transaction immediately visible to other transactions and users even before they are committed. This is the only possibility for local databases, which have very limited transaction support.

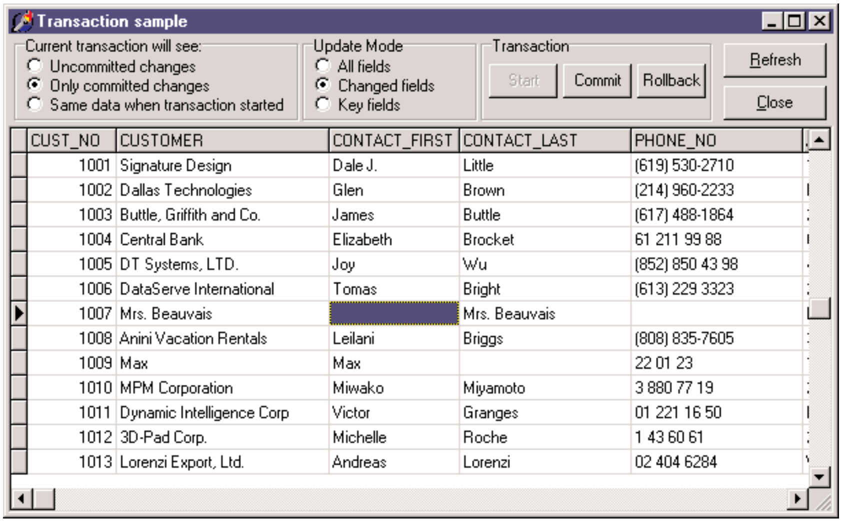
tiReadCommitted makes available to other transactions only the updates already committed.

tiRepeatableRead hides every other transaction started by other users after the current one. Following repeat calls within a transaction will always produce the same result, as if the database took a snapshot of the data when the current transaction started.

Most but not all SQL servers support only the most advanced levels. The default choice should be tiReadCommitted, which is quite powerful but not too heavy on the SQL server (as it adds very few internal locks).

As a general suggestion, transactions should involve only a minimal number of updates (only those strictly related and part of a single atomic operation) and should be kept short in time. You should avoid transactions that wait for user input to complete them, as the user might be temporarily gone and the transaction might remain active for a long time. Using `update` statements on multiple records and using cached updates can help us make the transactions small and fast.

To further inspect transactions and experiment with the update mode of the Table component, you can use the TranSample application. As you can see in Figure 14.3, you can simply use the radio buttons to choose the alternatives, and click the push buttons on the right of the toolbar to manually start, commit, and roll back a transaction. To get a real idea of the different effects, you should run multiple copies of the program (provided you have enough licenses on your InterBase server).



Current transaction will see:
○ Uncommitted changes
● Only committed changes
○ Same data when transaction started

Update Mode
○ All fields
● Changed fields
○ Key fields

Transaction

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO
1001	Signature Design	Dale J.	Little	(619) 530-2710
1002	Dallas Technologies	Glen	Brown	(214) 960-2233
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864
1004	Central Bank	Elizabeth	Brocket	61 211 99 88
1005	DT Systems, LTD.	Joy	Wu	(852) 850 43 98
1006	DataServe International	Tomas	Bright	(613) 229 3323
1007	Mrs. Beauvais		Mrs. Beauvais	
1008	Anini Vacation Rentals	Leilani	Briggs	(808) 835-7605
1009	Max	Max		22 01 23
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61
1013	Lorenzi Export, Ltd.	Andreas	Lorenzi	02 404 6284

FIGURE 14.3:

The TranSample application allows you to test the transaction isolation of a database and the update modes of a table.

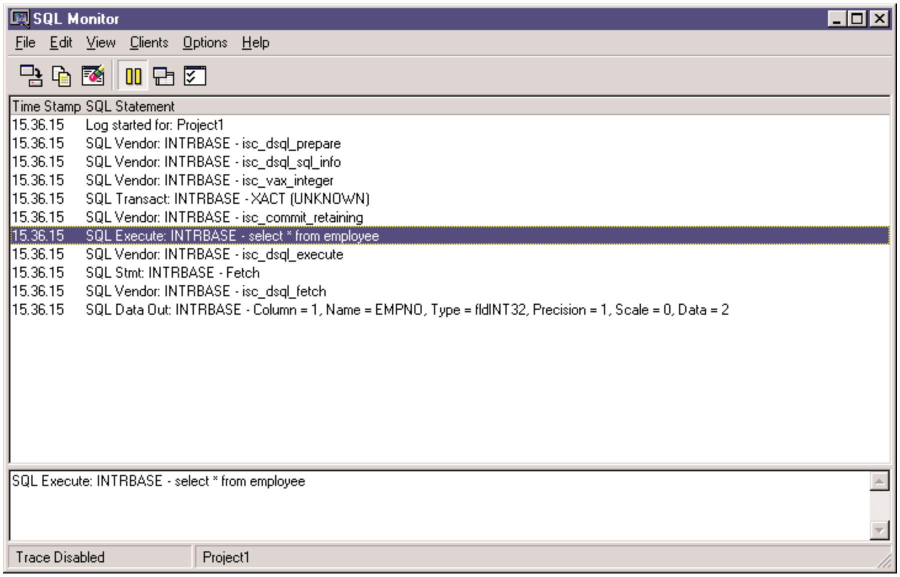
Using SQL Monitor

Just as you need a debugger to test a Delphi application, you need some tools to test how a client/server application behaves and to speed it up if possible. In particular, it is very important to look at the information moving from the client to the server (the explicit SQL requests our program does and those added by the BDE) and from the server to the client (the actual data). This is what the SQL Monitor tool included in Delphi Enterprise is for.

NOTE Notice that the SQL Monitor tool is specific to the BDE. The dbExpress and InterBase Express component sets have similar monitoring capabilities, directly embedded inside specific components.

As you can see in Figure 14.4, the central window of SQL Monitor shows a list of the low-level commands sent to the server. The bottom portion of the window shows the selected line of the above list on multiple rows, which helps when the line is too long.

FIGURE 14.4: The SQL Monitor running



```
Time Stamp SQL Statement
15.36.15   Log started for: Project1
15.36.15   SQL Vendor: INTRBASE - isc_dsql_prepare
15.36.15   SQL Vendor: INTRBASE - isc_dsql_sql_info
15.36.15   SQL Vendor: INTRBASE - isc_vax_integer
15.36.15   SQL Transact: INTRBASE - XACT (UNKNOWN)
15.36.15   SQL Vendor: INTRBASE - isc_commit_retaining
15.36.15   SQL Execute: INTRBASE - select * from employee
15.36.15   SQL Vendor: INTRBASE - isc_dsql_execute
15.36.15   SQL Stmt: INTRBASE - Fetch
15.36.15   SQL Vendor: INTRBASE - isc_dsql_fetch
15.36.15   SQL Data Out: INTRBASE - Column = 1, Name = EMPNO, Type = fldINT32, Precision = 1, Scale = 0, Data = 2
```

```
SQL Execute: INTRBASE - select * from employee
```

Trace Disabled | Project1

To use SQL Monitor, simply select the client program you want to inspect. Then set the proper trace options (by using the corresponding speed button or the Options ➢ Trace Options command). The available options are listed in Table 14.1.

TABLE 14.1: The Trace Options of the SQL Monitor

Trace Option	Meaning
Prepared Query Statement	Enables tracing of the SQL statements every time they are prepared.
Executed Query Statement	Traces all the SQL statements sent to the server.
Input Parameters	Shows input parameters as they become available. This is important for testing whether the parameters are correct.
Fetched Data	Shows the data sent by the server (a very slow operation).
Statement Operations	Shows the requests preceding the execution of a SQL statement, such as the allocation, preparation, and parsing of the input.
Connect/Disconnect	Shows the connection and disconnection events. This is an important test when the `KeepConnection` of the `Database` component is set to False, as the client won't maintain the connection with the server but will establish it only as needed (with the side effect of reducing the number of licenses required, in some cases). Looking at the frequency of these events might help you understand whether it is better to keep the connection active or not.
Transactions	Traces the transactions, including those activated automatically by the BDE if you don't use transactions directly.
Blob I/O	Shows the data about BLOB fields.
Miscellaneous	Traces other operations that don't fit any of the above categories.
Vendor Errors	Shows server error messages.
Vendor Calls	Shows client API calls.

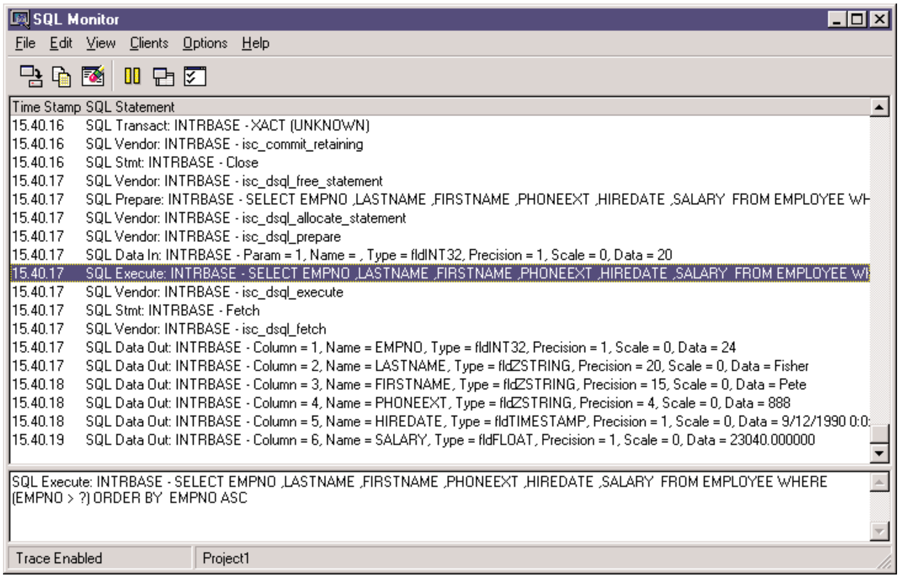
SQL Monitor is useful for seeing if the SQL statements sent by the BDE to the server are correct, but it also helps you see how many operations are done behind the scenes. Along with the time-stamp information for each operation, the number of operations can give some clue about your application's speed (although you should remember that the presence of SQL Monitor slows down the connection quite a lot).

In other words, SQL Monitor should be your guide in determining how to speed up your client/server application, using some of the tricks described in this chapter. At the same time, however, it takes a lot of experience and a good understanding of SQL to interpret its output properly.

As an example of the use of SQL Monitor, we can test what happens when we use the `Filter` property of a Table component. In a new project, simply place a Table, a DataSource, and a DBGrid. Select a database and a table (for example, the EMPLOYEE table of IBLocal) and set the `Filtered` property to True and the `Filter` property to `EmpNo>20`. If you now run the program, SQL Monitor will show you that the `select` statement generated by the BDE has a `where` clause corresponding to the filter. You can see this situation in Figure 14.5.

FIGURE 14.5:

SQL Monitor showing SQL statements generated by a Table component



```
Time Stamp SQL Statement
15.40.16    SQL Transact: INTRBASE - XACT (UNKNOWN)
15.40.16    SQL Vendor: INTRBASE - isc_commit_retaining
15.40.16    SQL Stmt: INTRBASE - Close
15.40.17    SQL Vendor: INTRBASE - isc_dsql_free_statement
15.40.17    SQL Prepare: INTRBASE - SELECT EMPNO ,LASTNAME ,FIRSTNAME ,PHONEEXT ,HIREDATE ,SALARY  FROM EMPLOYEE WH
15.40.17    SQL Vendor: INTRBASE - isc_dsql_allocate_statement
15.40.17    SQL Vendor: INTRBASE - isc_dsql_prepare
15.40.17    SQL Data In: INTRBASE - Param = 1, Name = , Type = fldINT32, Precision = 1, Scale = 0, Data = 20
15.40.17    SQL Execute: INTRBASE - SELECT EMPNO ,LASTNAME ,FIRSTNAME ,PHONEEXT ,HIREDATE ,SALARY  FROM EMPLOYEE WH
15.40.17    SQL Vendor: INTRBASE - isc_dsql_execute
15.40.17    SQL Stmt: INTRBASE - Fetch
15.40.17    SQL Vendor: INTRBASE - isc_dsql_fetch
15.40.17    SQL Data Out: INTRBASE - Column = 1, Name = EMPNO, Type = fldINT32, Precision = 1, Scale = 0, Data = 24
15.40.17    SQL Data Out: INTRBASE - Column = 2, Name = LASTNAME, Type = fldZSTRING, Precision = 20, Scale = 0, Data = Fisher
15.40.18    SQL Data Out: INTRBASE - Column = 3, Name = FIRSTNAME, Type = fldZSTRING, Precision = 15, Scale = 0, Data = Pete
15.40.18    SQL Data Out: INTRBASE - Column = 4, Name = PHONEEXT, Type = fldZSTRING, Precision = 4, Scale = 0, Data = 888
15.40.18    SQL Data Out: INTRBASE - Column = 5, Name = HIREDATE, Type = fldTIMESTAMP, Precision = 1, Scale = 0, Data = 9/12/1990 0:0:
15.40.19    SQL Data Out: INTRBASE - Column = 6, Name = SALARY, Type = fldFLOAT, Precision = 1, Scale = 0, Data = 23040.000000

SQL Execute: INTRBASE - SELECT EMPNO ,LASTNAME ,FIRSTNAME ,PHONEEXT ,HIREDATE ,SALARY  FROM EMPLOYEE WHERE
(EMPNO > ?) ORDER BY  EMPNO ASC

Trace Enabled    Project1
```

Performance Tuning

Besides using the SQL Monitor (or another monitoring technique) to determine the potential bottlenecks in your applications, you can do several things to speed up your client/server programs. The key element to keep in mind—as I've stressed many times in this chapter—is to reduce the network traffic, by reducing the result sets returned by the server both in the number of records and in the size of each.

Besides a good overall database design and a good Delphi implementation of it, there are many settings you can check. The following tips might come in handy, but they won't help as much as a better design!

- In InterBase, you can set an automatic sweep (or "garbage collection") interval. The operation is also automatically performed when you do a backup. Because a sweep slows down the database, it should not be done too frequently. However, if you never do it, the database will keep track of many leftover deleted records, reducing the overall performance and using extra memory.
- Use indexes on the fields used more often, particularly if you sort the result set on them. Keep in mind, though, that a good RDBMS will add at least temporary indexes for you. Using indexes can speed up queries quite a lot, particularly if the indexed fields are used to join two tables.

Continued on next page

- If you sort a field in descending order, a corresponding descending index might help.
- If you're an expert user, you might examine the *query plan*, the approach used by the server to perform a query, which is displayed (for example) when you use WISQL. The query plan will show you whether the SQL server is using indexes. In some cases, you might need to modify some complex queries to help the query optimizer built into the RDBMS.
- Check the server settings, including its cache, to obtain the best overall performance. The operating system cache on the server computer might help as well. In InterBase, if you want to perform all the updates physically, you can set the Forced Writes option in the Maintenance ➢ Database Properties menu of the InterBase Server Manager.
- Whenever possible, avoid an excessive use of transactions and try to keep them short and focused. Use a local cache instead of transactions (or together with them) to let the client computer do some more work for you, and skip some costly server operations.
- Handle transactions directly, disabling the auto-commit feature of the BDE; to do this, set `SQLPASSTHRU MODE` to `SHARED NOAUTOCOMMIT`. (You can set this and other BDE features described in this list with the BDE Administrator program.)
- If you have no licensing problems, set the `KeepConnection` property of the BDE Database component to True.
- With the BDE, set `TRACE MODE` to `0` when you are not debugging, to avoid having the drivers send trace strings to the debugger and slowing down the operations. Also the other monitoring services should be disabled when you're not debugging the application.
- In the BDE, enable schema caching (set `ENABLE SCHEMA CACHE` to `TRUE`). This setting reduces the time required to open a table, as the client doesn't need to ask for the metadata. You can also use the Delphi `FieldDefs` and `StoreDefs` properties of the Table component to store the metadata directly in the client program.
- With Microsoft and Sybase SQL Servers, try to set the `PACKETSIZE` parameter to a minimum of 4 KB, also modifying the corresponding value on the server. With these servers, also check that the `DRIVER FLAGS` parameter is set to `0`. If it is `2048`, queries will be executed in asynchronous mode and will be much slower.
- With ORACLE, DB/2, and ODBC drivers, try to fine-tune the `ROWSET SIZE` parameter until you obtain the best performance.
- With the InterBase driver, if you don't use explicit transactions, set the `DRIVER FLAGS` parameter to `4096`. This value enables *soft commits*, meaning that after each commit or rollback operation the open cursors won't have to be refreshed.

The dbExpress Library

As I mentioned in Chapter 13, one of the most notable new features of Delphi 6 is the adoption of the dbExpress database access library, a new SQL server access layer introduced by Borland in its Kylix product for Linux, and now with Delphi 6 also for Windows. As I've already provided a general overview of dbExpress in the preceding chapter, let's focus right away on the technical details.

Working with Unidirectional Cursors

The motto of dbExpress could be "fetch but don't cache." The key difference between this library and BDE or ADO is that dbExpress can only execute SQL queries and fetch the results in a *unidirectional cursor*. In "unidirectional" database access, you can move from one record to the next, but you cannot get back to a previous record of the dataset. This is because the library doesn't store the data is has retrieved in a local cache, but only passes it from the database server to the calling application.

Using a unidirectional cursor might sound like a limitation, and it really is! Besides having problems with the navigation, you cannot connect a database grid to a dataset like this. So what is a unidirectional dataset good for?

- You can use a unidirectional dataset for reporting purposes. In a printed report, but also an HTML page or an XML transformation, you move from record to record, produce the output, and that's it. No need to get back to past records and, in general, no interaction of the user with the data. Unidirectional datasets are probably the best option for Web and multitier architectures.
- You can use a unidirectional dataset to feed a local cache, such as the one provided by a ClientDataSet component. At this point, you can connect visual components to the in-memory dataset and operate on it with all the standard techniques, including the use of visual grids. You can freely navigate and edit the data in the in-memory cache, but also control it far better than with the BDE or ADO.

The important thing to notice is that, in these circumstances, avoiding the caching of the database engine actually saves time and memory. The library doesn't have to use extra memory for the cache and doesn't need to waste time storing data, duplicating information. Over the last couple of years, many programmers moved from BDE-based cached updates to the ClientDataSet component, which provides more flexibility in managing the content of the data and update information they keep in memory. However, using a ClientDataSet on top of the BDE (or ADO) exposes you to the risk of having two separate caches, actually wasting a lot of memory.

Another advantage of using the ClientDataSet component is that its cache supports editing operations, and the updates stored in this cache can be applied to the original database server by the DatasetProvider component. This component can generate the proper SQL update statements, and can do so in a more flexible way than the BDE (although ADO is quite powerful in this respect). In general, the provider can also use a dataset for the updates, but this isn't directly possible with the dbExpress dataset components.

Platforms and Databases

A key element of the dbExpress library is its availability for both Windows and Linux, in contrast to all the other database engines available for Delphi (BDE and ADO), which are available only for Windows. Notice, though, that some of the database-specific components, such as InterBase Express, are also available on multiple platforms.

When you use dbExpress, you are provided with a common framework, which is independent from the actual SQL database server you are planning to use. dbExpress comes with drivers for MySQL, InterBase, Oracle, and IBM DB2. These drivers are available as separate DLLs you have to deploy along with your program or as compiled units you can link into the executable file.

NOTE It is actually possible to write custom drivers for the dbExpress architecture. This is documented in details in the paper *dbExpress Draft Specification*, published on the Borland Community Web site. At the time of this writing, this document is at `http://community.borland.com/article/0,1410,22495,00.html`. You'll probably be able to find third-party drivers. For example, there is a free one (available also in Kylix), which bridges dbExpress and ODBC.

The dbExpress Components

The VCL components used to interface the dbExpress library encompass a group of dataset components plus a few ancillary ones. To differentiate these components from other database-access families, the components are prefixed with the letters *SQL*, underlining the fact that they are used for accessing RDBMS servers.

These components include a database connection component, a few dataset components (a generic one, three specific versions for tables, queries, and stored procedures, and one encapsulating a ClientDataSet component), and a monitor utility.

The SQLConnection Component

The `TSQLConnection` class inherits from the `TCustomConnection` component, and handles database connections, the same as its sibling classes (the Database, ADOConnection, and IBConnection components).

TIP

Unlike other component families, in dbExpress the connection is compulsory. In each of the dataset components, you cannot specify directly which database to use, but can only refer to a SQLConnection.

The connection component uses the information available in the drivers.ini and connections.ini files, which are the only two configuration files of dbExpress (these files are saved by default under \Program Files\Common Files\Borland Shared\DBExpress). The first, drivers.ini, lists the available dbExpress drivers, one for each supported database. For each driver, there is a set of default connection parameters. For example, the InterBase section reads as follows:

```
[Interbase]
GetDriverFunc=getSQLDriverINTERBASE
LibraryName=dbexpint.dll
VendorLib=GDS32.DLL
BlobSize=32
CommitRetain=True
Database=database.gdb
Password=masterkey
RoleName=RoleName
TransIsolation=ReadCommited
User_Name=sysdba
WaitOnLocks=True
```

The parameters indicate the dbExpress driver DLL (the LibraryName value), the entry function to use (GetDriverFunc), the vendor client library, and some more specific parameters that depend on the database. If you read the entire drivers.ini file, you'll see that the parameters are really database-specific. I have to say that some of these parameters don't make a lot of sense at the driver level, such as the database to connect to, but the list includes all the available parameters, regardless of their actual usage.

The connections.ini file provides the database specific description. This list resembles the aliases of the BDE, and you can enter multiple connection details for every database driver. The connection describes the physical database you want to connect to. As an example, this is the portion for the default IBLocal definition:

```
[IBLocal]
BlobSize=32
CommitRetain=True
Database=database.gdb
DriverName=Interbase
Password=masterkey
RoleName=RoleName
TransIsolation=ReadCommited
User_Name=sysdba
WaitOnLocks=True
```

As you can see by comparing the two listings, this is a subset of the parameters of the driver. When you create a new connection, the system will copy the default parameters from the driver; you can then edit them for the specific connection—for example, providing a proper database name. Each connection relates to the driver for its key attributes, as indicated by the `DriverName` property.

The important thing to notice is that these initialization files are used only at design time. In fact, when you select a driver or a connection at design time, the values of these files are copied to corresponding properties of the SQLConnection component, as in this example:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  DriverName = 'Interbase'
  GetDriverFunc = 'getSQLDriverINTERBASE'
  LibraryName = 'dbexpint.dll'
  LoginPrompt = False
  Params.Strings = (
    'BlobSize=-1'
    'CommitRetain=False'
    'Database=c:\program files\interbase corp\interbase6\examples\' +
      'database\employee.gdb'
    'DriverName=Interbase'
    'LocaleCode=0x0000'
    'Password=masterkey'
    'RoleName=RoleName'
    'ServerCharSet=ASCII'
    'SQLDialect=1'
    'Interbase TransIsolation=ReadCommited'
    'User_Name=sysdba'
    'WaitOnLocks=True')
  VendorLib = 'GDS32.DLL'
end
```

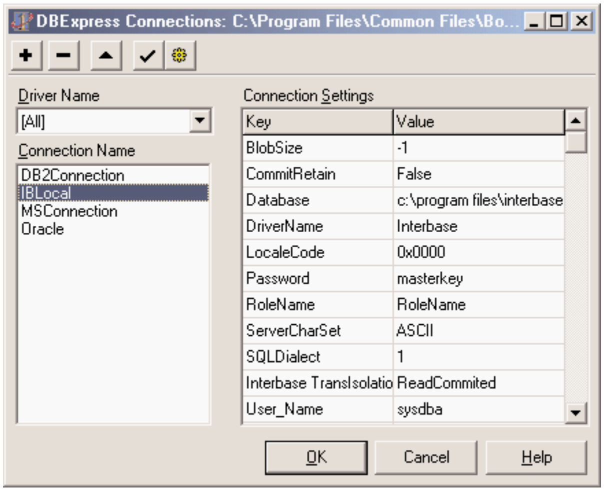
At run time, your program will rely on the properties to have all the required information, so you don't need to deploy the two INI files along with your programs. In theory, the files will be required if you want to change the `DriverName` or `ConnectionName` properties at run time. However, in case you want to connect your program to a new database, you can set directly the relevant properties.

When you add a new SQLConnection component to an application, you can proceed in different ways. You can set up a driver, using the list of values available for the `DriverName` property, and then select a predefined connection, by selecting one of the values available in the `ConnectionName` property. This second list is filtered according with the driver you've already selected. As an alternative, you can start by selecting directly the `ConnectionName` property, which in this case includes the entire list.

Instead of hooking up an existing connection, you can define a new one (or see the details of the existing connections) by double-clicking the SQLConnection component and launching the dbExpress Connection Editor (Figure 14.6). This editor lists, on the left side, all of the predefined connections, for a specific driver or all of them, and allows you to edit the connection properties using the grid on the right. You can use the toolbar buttons to add, delete, rename, and test connections, and to open the read-only dbExpress Drivers Settings window, shown in Figure 14.7.

FIGURE 14.6:

The dbExpress Connection Editor



DBExpress Connections: C:\Program Files\Common Files\Bo...

Driver Name: [All]

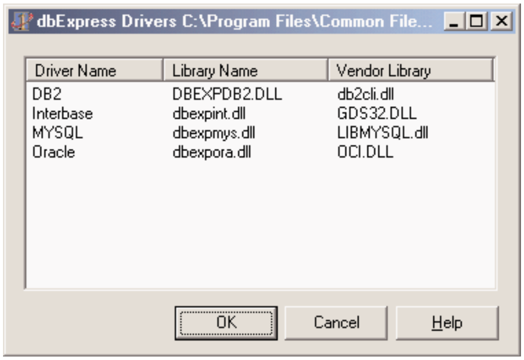
Connection Name:
DB2Connection
IBLocal
MSConnection
Oracle

Connection Settings

Key	Value
BlobSize	-1
CommitRetain	False
Database	c:\program files\interbase
DriverName	Interbase
LocaleCode	0x0000
Password	masterkey
RoleName	RoleName
ServerCharSet	ASCII
SQLDialect	1
Interbase TransIsolatio	ReadCommited
User_Name	sysdba

FIGURE 14.7:

The dbExpress Drivers Settings window of the dbExpress Connection Editor



dbExpress Drivers C:\Program Files\Common File...

Driver Name	Library Name	Vendor Library
DB2	DBEXPDB2.DLL	db2cli.dll
Interbase	dbexpint.dll	GDS32.DLL
MYSQL	dbexpmys.dll	LIBMYSQL.dll
Oracle	dbexpora.dll	OCI.DLL

Besides editing the predefined connection settings, the dbExpress Connection Editor allows you also to select a connection for the SQLConnection component. This is what the OK button is for. Notice, in fact, that if you change some of the settings, the data is immediately written to the INI files: clicking the Cancel button doesn't revert your editing!

If you want to define access to a database, editing the connection properties is certainly the suggested approach. This way when you need to access the same database from another application, or another connection within the same application, all you need to do is to select the connection. However, since this operation copies the connection data, notice that updating the connection doesn't automatically refresh the values within other SQLConnection components referring to the same named connection: you have to reselect the connection these other components refer to. In this respect, the predefined connections are very different from the BDE aliases.

What really matters for the SQLConnection component is the value of its properties. Driver and vendor libraries are listed in properties you can freely change at design time (although you'll rarely want to do this), while the database and other database-specific connection settings are specified in the `Params` properties. This is a string list including information such as the database name, the user name and password, and so on. In practice, you could set up a SQL-Connection component by setting up the driver and then assigning the database name directly in the `Params` property, forgetting about the predefined connection. I'm not suggesting this as the best option, but it is certainly a possibility; the predefined connections are handy, but when the data changes, you still have to manually refresh every SQLConnection component.

Actually, to be complete, I have to mention that there is an alternative. You can set the `LoadParamsOnConnect` property to indicate that you want to refresh the component parameters from the initialization files every time you open the connection. In this case, a change in the predefined connections will be reloaded when you open the connection, at either design time or run time. At design time, this provides a handy technique (which has the same effect as reselecting the connection), but using it at run time means you'll also have to deploy the `connections.ini` file, which can be a good idea or an inconvenient one, depending on your deployment environment.

The only property of the SQLConnection component that is not related to the driver and database settings is `LoginPrompt`. Setting it to False allows you to provide a password skipping the login request, both at design time and run time. If this is very handy for development, it can reduce the security of your system. Of course this is also the option you'll want to use for unattended connections, for example on a Web server.

The dbExpress Dataset Components

The dbExpress component's family provides four different dataset components: a generic dataset, a table, a query, and a stored procedure. The latter three components are provided for compatibility with the equivalent BDE components and have similarly named properties. If you don't have to port existing code, you should tend to use the general SQLDataSet component, which can be used to execute a query but also to access a table or a stored procedure.

The first important thing to notice is that all of these datasets inherit from a new special base class, `TCustomSQLDataSet`. This class and its derived classes represent unidirectional datasets, with the key features I've already described. In practice, this means that the browse operations are limited to calling `First` and `Next`, while `Prior`, `Last`, `Locate`, the use of bookmarks, and all other navigational features are disabled.

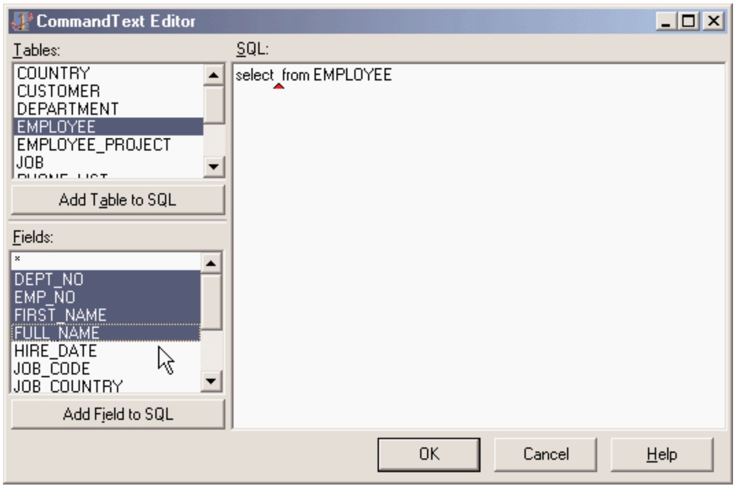
NOTE Technically, some of the moving operations call the `CheckBiDirectional` internal function and eventually raise an exception. `CheckBiDirectional` refers to the public `IsUnidirectional` property of the `TDataSet` class, which you can eventually use in your own code to disable operations that are illegal on unidirectional datasets.

Besides having limited navigational capabilities, these datasets have no editing support, so a lot of methods and events common to other datasets are not available. For example, there is no `AfterEdit` or `BeforePost` event.

As I mentioned earlier, of the four dataset components for dbExpress, the fundamental one is `TSQLDataSet`, which can be used both to retrieve a dataset and to execute a command. The two alternatives are activated by calling the `Open` method (or setting the `Active` property to True) and by calling the `ExecSQL` method.

The SQLDataSet component can retrieve an entire table, or use a SQL query or a stored procedure for reading a dataset or issuing a command. The `CommandType` property determines one of the three access modes. The possible values are ctQuery, ctStoredProc, and ctTable, which determine the value of the `CommandText` property (and also the behavior of the related property editor in the Object Inspector). For a table or stored procedure, the `CommandText` property indicates the name of the related element of the database, and the editor provides a drop-down list with the possible values. For a query, the `CommandText` property stores the text of the SQL command, and the editor provides a little help in building the SQL query (in case it is a select statement). You can see the editor in Figure 14.8.

When you use a table, the component will generate a SQL query for you, as dbExpress targets only SQL databases. The generated query will include all the fields of the table, and if you specify the `SortFieldNames` property, it will include a `sort by` directive.



```
select from EMPLOYEE
```

FIGURE 14.8: The CommandText Editor used by the SQLDataSet component for queries

The three *specific* dataset components offer a similar behavior, but you specify the SQL query in the `SQL` string list property, the stored procedure in the `StoredProcName` property, and the table name in the `TableName` property, as in the three corresponding BDE components.

The SQLClientDataSet Component

The SQLClientDataSet is a combination of three components: the SQLDataSet component, a (hidden) provider, and the ClientDataSet. The idea is to be a helper, as you need only one component instead of three, which must also be connected. However, it doesn't surface all of the properties and events of the underlying components, so in complex situations, it's better to use the various components it stands for. I'll cover this and other variations of the ClientDataSet component later in this chapter, after I discuss the ClientDataSet itself in detail.

The SQLMonitor Component

The final component of the dbExpress group is the SQLMonitor, used to log the requests sent from dbExpress to the database server. This monitor provides capabilities similar to the stand-alone SQL Monitor application, which is bound to the BDE and cannot be used with the dbExpress library (and the analogous IBXMonitor component of the InterBase Express family, as we'll see in the next chapter).

The TimeStamp Field Type

Along with dbExpress, Delphi 6 introduces the `TSQLTimeStampField` field type, mapped to the timestamp data type that many SQL servers have (InterBase included). This data type is now available also in Delphi and is called `TSQLTimeStamp`. A time stamp is a simple record-based representation of a date or time, quite different from the floating-point representation used by the `TDateTime` data type. A time stamp is defined as:

```
TSQLTimeStamp = packed record
  Year : SmallInt;
  Month : Word;
  Day : Word;
  Hour : Word;
  Minute : Word;
  Second : Word;
  Fractions : LongWord;
end;
```

A time stamp field can automatically covert standard date and time values using the `AsDateTime` property (as opposed to the native `AsSQLTimeStamp` property). You can also do custom conversions and further manipulation of time stamps by using the routines provided by the SqlTimSt unit, including functions like `DateTimeToSQLTimeStamp`, `SQLTimeStampToStr`, and `VarSQLTimeStampCreate`.

A Simple dbExpress Demo

After that introduction, let's have a look at an actual demonstration, highlighting the key features of these components and showing how to use the ClientDataSet to provide caching and editing support for the unidirectional datasets. In another example, later on, I'll show you an example of native use of the unidirectional query, with no caching and editing support required.

The standard visual application based on dbExpress uses this series of components:

- The SQLConnection component provides the connection with the database and the proper dbExpress driver.
- The SQLDataSet component, which is hooked to the connection (via the `SQLConnection` property), indicates which SQL query to execute or table to open (using the `CommandType` and `CommandText` properties discussed earlier).
- The DataSetProvider component, connected with the dataset, extracts the data from the SQLDataSet and can generate the proper SQL update statements.

- The ClientDataSet component reads from the data provider and stores all the data (if its PacketRecords property is set to –1) in memory. This component has a lot of extra features and provides the actual data to the application, with full navigation and editing capabilities. You'll need at least to call its ApplyUpdates method to send the actual updates back to the database server (through the provider).
- The DataSource component allows you to surface the data from the ClientDataSet to the visual data-aware controls.

As I mentioned earlier, the picture can be simplified by using the SQLClientDataSet, which replaces the two datasets and the provider. The SQLClientDataSet combines most of the properties of the components it replaces. For a simple example, you'll have to set the DBConnection property for connecting to the proper database, the CommandType and CommandText properties to specify which data to fetch, and the PacketRecords property to indicate how many records to retrieve in each block. You'll also need to call to the ApplyUpdates method to send the actual updates back to the database.

These are the key properties of the core components of the DbxSingle example:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLClientDataSet1: TSQLClientDataSet
  CommandText = 'EMPLOYEE'
  CommandType = ctTable
  DBConnection = SQLConnection1
end
```

As an alternative, the DbxMulti example uses the entire sequence of components:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText = 'select * from EMPLOYEE'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'DataSetProvider1'
end
object DataSource1: TDataSource
  DataSet = ClientDataSet1
end
```

Both examples also have some visual controls: a grid and a toolbar based on the action manager architecture.

Applying Updates

What is important to do in every example based on a local cache, like the one provided by the ClientDataSet and SQLClientDataSet components, is to write the local changes back to the database server. This is typically accomplished by calling the ApplyUpdates method. But when should you call it? You can either keep the changes in the local cache for a while and then apply a bunch of updates at once, or post each change right away. In these two simple examples, I've gone for the latter approach, attaching the following event handler to the AfterPost (fired after an edit or an insert operation) and AfterDelete events of the ClientDataSet components:

```
procedure TForm1.DoUpdate(DataSet: TDataSet);
begin
  // immediately apply local changes to the database
  SQLClientDataSet1.ApplyUpdates(0);
end;
```

If you want to apply all the updates in a single batch, you can either do this when the form is closed or the program ends, or let a user do the update operation by selecting a specific command. We'll explore some of these alternatives when discussing the ClientDataSet component in more detail.

Monitoring the Connection

Another feature, which I've added only to the DbxSingle example, is the monitoring capability offered by the SQLMonitor component. In the example, the component is activated as the program starts.

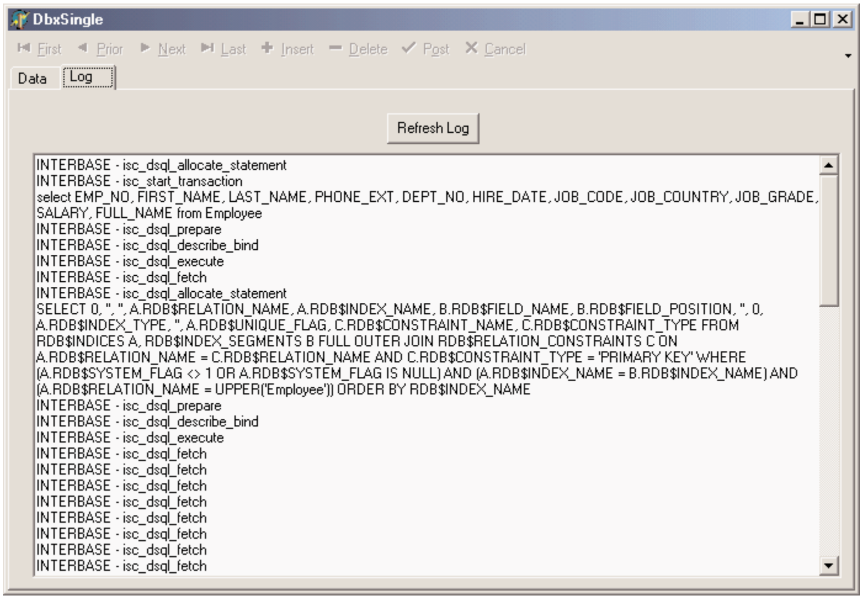
Every time there is a tracing string available, the component fires the OnTrace event to let you choose whether to include the string in the log. If the LogTrace parameter of this event is True (the default value), the component logs the message in the TraceList string list and fires the OnLogTrace event to indicate that a new string has been added to the log.

The component can also automatically store the log into the file indicated by its FileName property, but I haven't used this feature in the example. All I've done is to handle the OnLogTrace event, adding the last message to a memo component with the following code (and the output of Figure 14.9):

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject;
  CBInfo: pSQLTRACEDesc);
begin
  MemoLog.Lines.Add (CBInfo.pszTrace);
end;
```

FIGURE 14.9:

A sample log obtained by the SQLMonitor in the DbxSingle example.



```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, DEPT_NO, HIRE_DATE, JOB_CODE, JOB_COUNTRY, JOB_GRADE,
SALARY, FULL_NAME from Employee
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME, B.RDB$FIELD_NAME, B.RDB$FIELD_POSITION, '', 0,
A.RDB$INDEX_TYPE, '', A.RDB$UNIQUE_FLAG, C.RDB$CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE FROM
RDB$INDICES A, RDB$INDEX_SEGMENTS B FULL OUTER JOIN RDB$RELATION_CONSTRAINTS C ON
A.RDB$RELATION_NAME = C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY' WHERE
(A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG IS NULL) AND (A.RDB$INDEX_NAME = B.RDB$INDEX_NAME) AND
(A.RDB$RELATION_NAME = UPPER('Employee')) ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
```

Also, a button allows a user to refresh the memo with the entire content of the trace list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MemoLog.Lines := SQLMonitor1.TraceList;
end;
```

Controlling the SQL Update Code

If you run the DbxSingle program and change, for example, the telephone number of an employee, the monitor will log this update operation:

```
update EMPLOYEE  set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  FIRST_NAME = ? and
  LAST_NAME = ? and
  PHONE_EXT = ? and
  HIRE_DATE = ? and
  DEPT_NO = ? and
  JOB_CODE = ? and
  JOB_GRADE = ? and
  JOB_COUNTRY = ? and
  SALARY = ? and
  FULL_NAME = ?
```

The structure of the update statement depends on the UpdateMode property of the SQLClientDataSet component. Trying to use upWhereChanged or upWhereKeyOnly, however, causes an error, as the component is unable to determine which are the key records, and the provider doesn't generate a correct update statement. In fact, it simply tries to update the update the record based on the specific field changed, without including the key field in the where statement:

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  PHONE_EXT = ?
```

The update statement should rather be:

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  PHONE_EXT = ?
```

How can we obtain the proper update call? Considering we cannot directly attach a component like the UpdateSQL (which is a BDE-only component), a simple solution would be to force the inclusion of the key field. This can be accomplished in the ClientDataSet architecture by turning on the pfInKey flag of the ProviderOptions property of the source field.

This can easily be accomplished in the DbxMulti example, after adding persistent fields for the SQLDataSet component, but the problem is that the database library should locate the key fields automatically. In the DbxSingle example, we have no control on the source dataset and its fields, so the only solution is to write the update statements in a totally custom way, with quite some effort.

NOTE We'll be able to discuss this type of problem again as we get into the details of the ClientDataSet component, the Provider, the Resolver, and other technical details later in this chapter and in Chapter 17.

Accessing Database Metadata with *SetSchemaInfo*

All RDBMS systems use special-purpose tables (generally called *system tables*) for storing metadata, such as the list of the tables, their fields, indexes, and constraints, and any other system information. As dbExpress provides a unified API for working with different SQL servers, it provides also a common way for accessing metadata. The TSQLDataSet component has a method, SetSchemaInfo, which fills the dataset with system information. This SetSchemaInfo method has three parameters:

SchemaType indicates the type of information requested and includes stTables, stSysTables, stProcedures, stColumns, and stProcedureParams.

SchemaObject indicates the object you are referring to, such as the name of the table for which you are requesting the columns.

SchemaPattern is a filter, so that you can limit your request to tables, columns, or procedures starting with the given letters. This is very handy if you use prefixes to identify groups of elements.

For example, in the SchemaTest program, a button reads into the dataset all of the tables of the connected database:

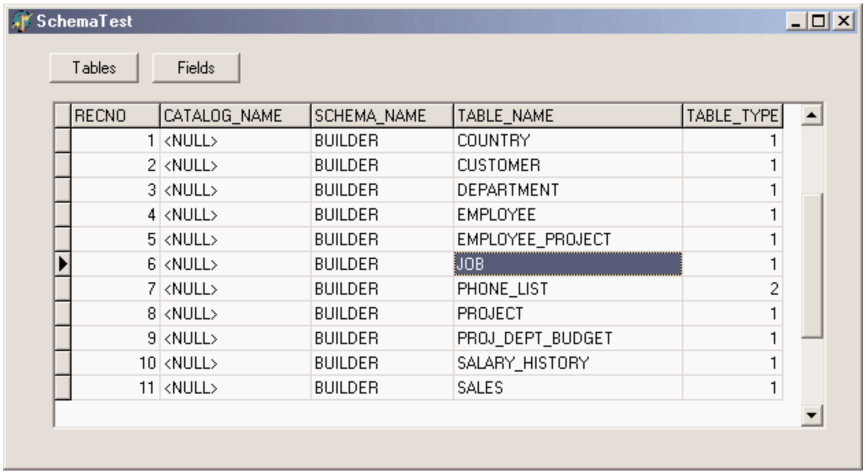
```
ClientDataSet1.Close;
SQLDataSet1.SetSchemaInfo (stTables, '', '');
ClientDataSet1.Open;
```

The program uses the usual group of dataset provider, client dataset and data source component to display the resulting data in a grid, as you can see in Figure 14.10. After you've retrieved the tables, you can select a row of the grid and press the second button to see a list of the fields of this table:

```
SQLDataSet1.SetSchemaInfo (stColumns, ClientDataSet1['Table_Name'], '');
ClientDataSet1.Close;
ClientDataSet1.Open;
```

FIGURE 14.10:

The SchemaTest example allows you to see the tables of a database and the columns of a given table.



RECNO	CATALOG_NAME	SCHEMA_NAME	TABLE_NAME	TABLE_TYPE
1	<NULL>	BUILDER	COUNTRY	1
2	<NULL>	BUILDER	CUSTOMER	1
3	<NULL>	BUILDER	DEPARTMENT	1
4	<NULL>	BUILDER	EMPLOYEE	1
5	<NULL>	BUILDER	EMPLOYEE_PROJECT	1
6	<NULL>	BUILDER	JOB	1
7	<NULL>	BUILDER	PHONE_LIST	2
8	<NULL>	BUILDER	PROJECT	1
9	<NULL>	BUILDER	PROJ_DEPT_BUDGET	1
10	<NULL>	BUILDER	SALARY_HISTORY	1
11	<NULL>	BUILDER	SALES	1

Besides accessing database metadata, dbExpress provides a way to access to its own configuration information, including the installed drivers and the configured connections. The unit DbConnAdmin defines a `TConnectionAdmin` class for this purpose, but the aim of this support is probably limited to dbExpress add-on utilities for developers, as letting end users access multiple databases in a totally dynamic way is not very common.

TIP The DbxExplorer demo included in Delphi 6 shows how to access both dbExpress administration files and schema information. Also check the help file under "The structure of metadata datasets" within the section "Developing database applications."

A Round-Up on dbExpress

After we've delved a little more into the dbExpress architecture and SQL components, I can try to add a few comments about this solution and its alternatives. On the whole, dbExpress provides a much neater architecture, compared to the BDE. In particular, I like it a lot when a database engine does things for me but lets me control what's going on and fine-tune every element. Ready-to-use defaults, such as those provided by the SQLClientDataSet component, are nice, but it is important for me in real-world applications to be able to take full control and write the exact SQL code I want my system to execute.

NOTE Another complaint I have is that I don't really like the architecture used for the specific ClientDataSet components bound to the various technologies (BDE, ADO, dbExpress and so on). In Chapter 19, "COM Programming," I'll discuss an alternate approach. In particular, I dislike the fact that you cannot code for a generic base component, but have to tune your code to the specific version of the ClientDataSet. Moreover, this architecture isn't extensible: you'll have to write a new specific component for each data access class you want to use (or to write). This seems really contrary to the spirit of OOP and to the overall architecture of VCL. My suggestion is simply to avoid using these all-in-one components and get used to dropping the DataSet–Provider–ClientDataSet triad every time you need to (or build a custom compound component for them).

When One-Way Is Enough: Printing Data

We have seen that one of the key elements of the dbExpress library is that it returns unidirectional datasets and that we can use the ClientDataSet component (in one of its incarnations) to store the records in a local cache. Now it is interesting to discuss at least a simple example where a unidirectional dataset is all we need.

This is common in *reporting*, that is, to produce information for each record in sequence without needing any further access to the data. This broad category includes producing printed reports (via a set of reporting components or using the printer directly), sending data to other applications like Microsoft Excel or Word, saving data to files (including HTML and XML formats), and more.

As I don't want to delve into HTML and XML right now, and we still haven't discussed COM-based automation, I'll go ahead with an example of printing—nothing fancy and nothing based on reporting components, but a simple way to produce a draft report on your video and printer. For this reason, I'm going to use Delphi's simplest technique to produce a printout: assigning a file to the printer with the AssignPrn RTL procedure.

The example, called UniPrint, has a unidirectional SQLDataSet component, hooked to an InterBase connection and based on the following SQL statement, which joins the employee table with the department table to display the name of the department where each employee works:

```
select d.DEPARTMENT, e.FULL_NAME, e.JOB_COUNTRY, e.HIRE_DATE
from EMPLOYEE e
inner join DEPARTMENT d on d.DEPT_NO = e.DEPT_NO
```

To handle printing, I've written a somewhat generic routine, requiring as parameters the data to print, a progress bar for status information, the output font, and the maximum format size of each field. The entire routine, listed below, uses file-print support and the graphic objects recall technique, and formats each field in a fixed-size, left-aligned string, to produce a columnar type of report. The call to the Format function has a parametric format string, built dynamically using the size of the field.

Here is the code, which uses three nested try/finally blocks to release all the resources properly:

```
procedure PrintOutDataSet (data: TDataSet;
  progress: TProgressBar; Font: TFont; maxSize: Integer = 30);
var
  PrintFile: TextFile;
  I: Integer;
  sizeStr: string;
  oldFont: TFontRecall;
begin
  // assign the printer to a file
  AssignPrn (PrintFile);
  Rewrite (PrintFile);

  // set the font and keep the original one
  oldFont := TFontRecall.Create (Printer.Canvas.Font);
  try
    Printer.Canvas.Font := Font;
    try
      data.Open;
      try
        // print header (field names) in bold
        Printer.Canvas.Font.Style := [fsBold];
        for I := 0 to data.FieldCount - 1 do
        begin
          sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
          Write (PrintFile, Format ('%-' + sizeStr + 's',
            [data.Fields[i].FieldName]));
        end;
        Writeln (PrintFile);

        // for each record of the dataset
        Printer.Canvas.Font.Style := [];
        while not data.EOF do
        begin
          // print out each field of the record
          for I := 0 to data.FieldCount - 1 do
          begin
            sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
            Write (PrintFile, Format ('%-' + sizeStr + 's',
              [data.Fields[i].AsString]));
          end;
          Writeln (PrintFile);
          // advance ProgressBar
          progress.Position := progress.Position + 1;
          data.Next;
        end;
      finally
        // close the dataset
        data.Close;
      end;
    finally
      // reassign the original printer font
      oldFont.Free;
    end;
  finally
    // close the printer/file
    System.CloseFile (PrintFile);
  end;
end;
```

The program invokes this routine when the Print All button is clicked. The program executes a separate query, which returns the number of records of the employee table to set up the progress bar (the unidirectional dataset, in fact, has no way to know how many records it is going to retrieve until it has reached the last one). Then it sets the output font, possibly using a fixed-width font, and calls the `PrintOutDataSet` routine.

```
procedure TNavigator.PrintAllButtonClick(Sender: TObject);
var
  Font: TFont;
begin
  // set ProgressBar range
  EmplCountData.Open;
  try
```

```
      ProgressBar1.Max := EmplCountData.Fields[0].AsInteger;
    finally
      EmplCountData.Close;
    end;

    Font := TFont.Create;
    try
      Font.Name := 'Courier New';
      Font.Size := 9;
      PrintOutDataSet (EmplData, ProgressBar1, Font);
    finally
      Font.Free;
    end;
  end;
```

ClientDataSet and MyBase

The general idea of a client/server application implies that the computation workload is shared between two separate programs, the RDBMS and a client application. Although it is very hard to strike a precise line between the two sides, it is certainly useful to do operations on the client. Most database engines (BDE, as we've seen in this chapter, and ADO, as we'll see in Chapter 16, "ActiveX Data Objects") can manipulate client-side data stored in a cache. Using the ClientDataSet component, you can do the same regardless of the database engine you are using, which makes your program more flexible, particularly if you want to use dbExpress, which doesn't provide a similar feature natively.

A practical example will underline what I mean: Suppose you've written a SQL query to retrieve a rather large dataset, and a user wants to see the same data in a different order. You can certainly run a new query, with the proper `order by` clause, but this implies sending the same (possibly large) dataset once more from the server to the client. Since the client already has the data in memory, it would be more practical and generally faster to re-sort the data in memory and present the same data to the user with a different ordering.

The ClientDataSet component allows you to do this: Attaching the code to sort the data by assigning a proper field name to the `IndexFieldNames` property. This is often accomplished when the user clicks the field title in a DBGrid component (firing the `OnTitleClick` event):

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  ClientDataSet1.IndexFieldNames := Column.Field.FieldName;
end;
```

TIP Unlike local databases, a ClientDataSet can have dynamic indexes, as they are computed in memory anyway. The component also supports indexes based on a calculated field, specifically an *internally calculated* field, a type of field available only for this dataset. Unlike ordinary calculated fields, which are computed every time the record is used, values of internally calculated fields are kept in memory. This is why indexes consider them as plain fields.

Indexing is not all the ClientDataSet has to offer. When you have an index, you can define groups based on it, possibly with multiple levels of grouping. There is even specific support for determining the position of a record within a group (first, last, or middle position). Over groups or entire tables, you can define aggregates; that is, you can compute the sum or average value of a column for the entire table or the current group on-the-fly. The data doesn't need to be posted to a physical server, because these aggregate operations take place in memory. You can even define new aggregate fields, to which you can directly connect data-aware controls. I'll explore these capabilities in the next section.

Another very interesting area of the ClientDataSet component is its ability to handle the updates log, undoing changes, looking at their list before committing them, and so on. I'll explore this next.

The ClientDataSet component supports many features, only some of which are related to the three-tier architecture (covered in Chapter 17). This component represents a database completely mapped in memory and can also be made persistent to a local file. Borland marketing has introduced the name MyBase to describe this feature of the ClientDataSet component, which was formerly called the briefcase model.

The important thing to keep in mind is that all of these features are available to any client/server and even local applications. The ClientDataSet component, in fact, can get its data from a remote connection, from a local dataset (as you must do with dbExpress), or from a local MyBase file. This is another huge area to explore, so I'll simply show you a couple of examples highlighting key features.

WARNING The use of the ClientDataSet component, in each of its incarnations, requires either the deployment of the `Midas.dll` library or the inclusion in the project of the MidasLib unit (available in compiled format only). The core code of this component, in fact, is not directly part of the library and is not available in source code format. This is unfortunate, as many Delphi developers are accustomed to debugging into the source code and using it as the ultimate reference. It is noteworthy, though, the inclusion in Delphi 6 of the DCU version of the library, obtained from a C-language source code. This allows you to avoid deploying the actual library along with your program.

The Packets and the Cache

The ClientDataSet component reads data in packets made of the number of records indicated by the PacketRecords property. The default value of this property is –1, which means that the provider will pull all the records at once (this is reasonable only for a small dataset). Alternatively, you can set this value to zero to ask the server for only the field descriptors and no actual data or use any positive value to specify an actual number.

If you retrieve only a partial dataset, as you browse past the end of local cache, if FetchOnDemand property is set to True (the default value), the ClientDataSet component will get more records from its source. This same property also controls whether BLOB fields and nested datasets of the current records are fetched automatically (these values might not be already part of the data packet, depending on the value of the Options of the dataset provider).

If you turn off this property, you'll need to manually fetch more records, by calling the GetNextPacket method, until the method returns zero. (You'll call FetchBlobs and FetchDetails for these other elements.)

WARNING Notice, by the way, that before you set a index for the data, you should retrieve the entire dataset (either by going to its last record or by setting the PacketRecords property to –1). Otherwise you'll have an odd index based on partial data.

Filtering

As with any other dataset, you can use the Filter property to specify the inclusion in the dataset of portions of the data the component is bound to. When manipulating a large table, of course, you should use a proper query so that you don't retrieve a large dataset from a SQL server. Filtering up-front in the server should generally be your first choice.

However, local filtering in the ClientDataSet can be quite useful, particularly because the filter expressions you can use are much more extensive than those you can use with other datasets. In particular, you can use the standard comparison and logical operators (`Population > 1000 and Area < 1000`) and arithmetic operators (`Population / Area < 10`), but also string functions (`Substring(Last_Name), 1, 2 = 'Ca'`), date and time functions (`Year (Invoice_Date) = 2002`), and others, including a Like function, wildcards, and an In operator.

These filtering capabilities are fully documented in the VCL Help file. Notice that the documentation was already there for Delphi 5, but most of these features didn't actually work. Now they do.

Grouping and Aggregates

We've already seen that a ClientDataSet can have an index different than the order in which it received the data. Once you've defined an index, you can group the data by that index. In practice, a group is defined as a list of consecutive records (according to the index) for which the value of the indexed field doesn't change. For example, if you have an index by state, all the addresses within that state will fall in the group.

Grouping

The CdsCalcs example has a ClientDataSet component that extracts its data from the Country table of the familiar DBDEMOS database. This operation is performed using a DataSetProvider component to the form, connecting the three components as follows:

```
object Table1: TTable
  Active = True
  DatabaseName = 'DBDEMOS'
  TableName = 'COUNTRY.DB'
end
object DataSetProvider1: TDataSetProvider
  DataSet = Table1
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'DataSetProvider1'
end
```

I could have also used the specific BDEClientDataSet component, as I'll discuss later. Now we can focus on the definition of the group. This is obtained, along with the definition of an index, by specifying a grouping level for the index itself:

```
object ClientDataSet1: TClientDataSet
  IndexDefs = <
    item
      Name = 'ClientDataSet1Index1'
      Fields = 'Continent'
      GroupingLevel = 1
    end>
  IndexName = 'ClientDataSet1Index1'
```

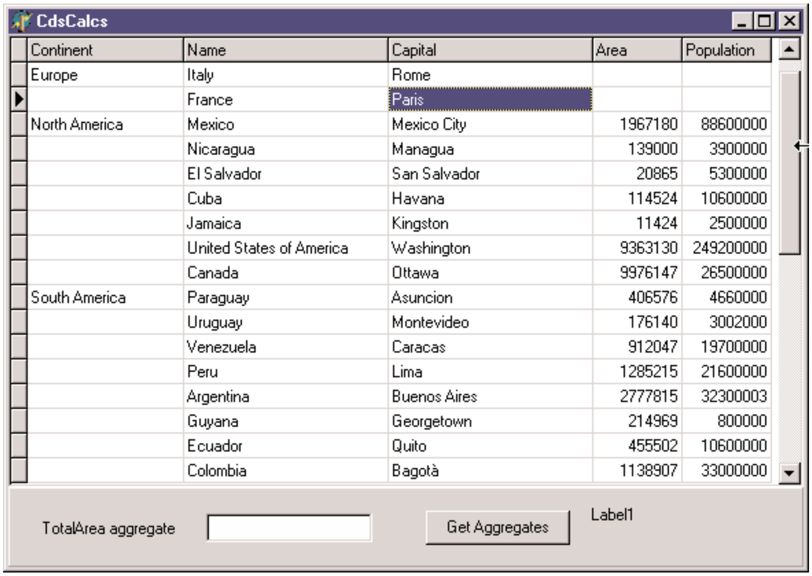
When you have a group active, you can make this obvious to the user by displaying the grouping structure in the DBGrid, as shown in Figure 14.11. Simply handle the `OnGetText` event for the grouped field (the `Continent` field in the example), and show the text only if the record is the first of the group:

```
procedure TForm1.ClientDataSet1ContinentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
```

```
    if gbFirst in ClientDataSet1.GetGroupState (1) then
      Text := Sender.AsString
    else
      Text := '';
  end;
```

FIGURE 14.11:

The CdsCalcs example demonstrates that by writing a little code, you can have the DBGrid control visually show the grouping defined in the ClientDataSet.



Continent	Name	Capital	Area	Population
Europe	Italy	Rome		
	France	Paris		
North America	Mexico	Mexico City	1967180	88600000
	Nicaragua	Managua	139000	3900000
	El Salvador	San Salvador	20865	5300000
	Cuba	Havana	114524	10600000
	Jamaica	Kingston	11424	2500000
	United States of America	Washington	9363130	249200000
	Canada	Ottawa	9976147	26500000
South America	Paraguay	Asuncion	406576	4660000
	Uruguay	Montevideo	176140	3002000
	Venezuela	Caracas	912047	19700000
	Peru	Lima	1285215	21600000
	Argentina	Buenos Aires	2777815	32300003
	Guyana	Georgetown	214969	800000
	Ecuador	Quito	455502	10600000
	Colombia	Bagotà	1138907	33000000

Defining Aggregates

Another feature of the ClientDataSet component is support for *aggregates*. An aggregate is a calculated value based on multiple records, such as the sum or the average value of a field for the entire table or a group of records (defined with the grouping logic I've just discussed). Aggregates are *maintained*; that is, they are recalculated immediately if one of the records changes. For example, the total of an invoice can be maintained automatically while the user types in the invoice items.

NOTE Aggregates are maintained incrementally, not by recalculating all the values every time one value changes. Aggregate updates take advantage of the deltas tracked by the ClientDataSet. For example, to update a sum when a field is changed, the ClientDataSet subtracts the old value from the aggregate and adds the new value. Only two calculations are needed, even if there are thousands of rows in that aggregate group. For this reason, aggregate updates are instantaneous.

There are two ways to define aggregates. You can use the Aggregates property of the ClientDataSet, which is a collection, or you can define aggregate fields using the Fields editor. In both cases, you define the aggregate expression, give it a name, and connect it to an index and a grouping level (unless you want to apply it to the entire table). Here is the Aggregates collection of the CdsCalcs example:

```
object ClientDataSet1: TClientDataSet
  Aggregates = <
    item
      Active = True
      AggregateName = 'Count'
      Expression = 'COUNT (NAME)'
      GroupingLevel = 1
      IndexName = 'ClientDataSet1Index1'
      Visible = False
    end
    item
      Active = True
      AggregateName = 'TotalPopulation'
      Expression = 'SUM (POPULATION)'
      Visible = False
    end>
  AggregatesActive = True
```

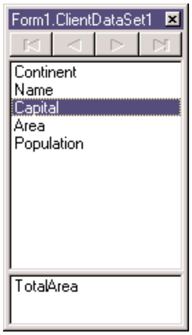
Notice in the last line above that you must activate the support for aggregates, in addition to activating each specific aggregate you want to use. Disabling aggregates is important, because having too many of them can slow down a program. The alternative approach, as I mentioned, is to use the Fields editor, select the New Field command of its shortcut menu, and choose the Aggregate option (available, along with the InternalCalc option, only in a ClientDataSet). This is the definition of an aggregate field:

```
object ClientDataSet1: TClientDataSet
  object ClientDataSet1TotalArea: TAggregateField
    FieldName = 'TotalArea'
    ReadOnly = True
    Visible = True
    Active = True
    DisplayFormat = '###,###,###'
    Expression = 'SUM(AREA)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
  end
```

The aggregate fields are displayed in the Fields editor in a separate group, as you can see in Figure 14.12. The advantage of using an aggregate field, compared to a plain aggregate, is that you can define the display format and hook the field directly to a data-aware control, such as a DBEdit in the CdsCalcs example. Because the aggregate is connected to a group, as soon as you select a record of a different group, the output will be automatically updated. Also, if you change the data, the total will immediately show the new value.

FIGURE 14.12:

The bottom portion of the Fields editor of a ClientDataSet displays aggregate fields.



Form1.ClientDataSet1

Continent
Name
Capital
Area
Population

TotalArea

To use plain aggregates, instead, you have to write a little code, as in the following example (notice that the `Value` of the aggregate is a variant):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption :=
    'Area: ' + ClientDataSet1TotalArea.DisplayText + #13'Population : '
    + FormatFloat ('###,###,###', ClientDataSet1.Aggregates [1].Value) +
    #13'Number : ' + IntToStr (ClientDataSet1.Aggregates [0].Value);
end;
```

Manipulating Updates

One of the core ideas behind the ClientDataSet component is that it is used as a local cache to collect some input from a user and then send a batch of update requests to the database. The component has both a list of the changes to apply to the database server, stored in the same format used by the ClientDataSet (accessible though the `Delta` property), and a complete updates log that you can manipulate with a few methods (including an Undo capability).

The Status of the Records

The component lets us monitor what's going on within the data packets. The `UpdateStatus` method returns one of the following indicators for the current record:

```
type TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

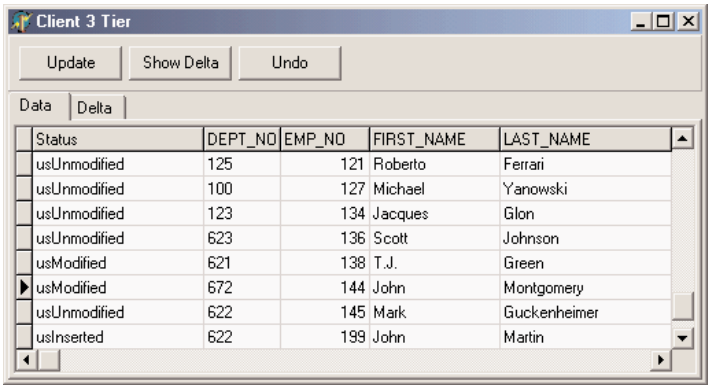
To check the status of every record in the client dataset easily, you can add a string-type calculated field to the dataset (I've called it `ClientDataSet1Status`) and compute its value with the following `OnCalcFields` event handler:

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  ClientDataSet1Status.AsString := GetEnumName (TypeInfo(TUpdateStatus),
    Integer (ClientDataSet1.UpdateStatus));
end;
```

This method (based on the RTTI `GetEnumName` function) converts the current value of the `TUpdateStatus` enumeration to a string, with the effect you can see in Figure 14.13.

FIGURE 14.13:

The CdsDelta program displays the status of each record of a ClientDataSet.



Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME
usUnmodified	125	121	Roberto	Ferrari
usUnmodified	100	127	Michael	Yanowski
usUnmodified	123	134	Jacques	Glon
usUnmodified	623	136	Scott	Johnson
usModified	621	138	T.J.	Green
usModified	672	144	John	Montgomery
usUnmodified	622	145	Mark	Guckenheimer
usInserted	622	199	John	Martin

Accessing the Delta

Beyond examining the status of each record, the best way to understand which changes have occurred in a given ClientDataSet (but haven't been uploaded to the server) is to look at the delta, the list of changes waiting to be applied to the server. This property is defined as follows:

```
property Delta: OleVariant;
```

The format used by the `Delta` property is the same as that used to transmit the data from the client to the server. What we can do, then, is add another ClientDataSet component to an application and connect it to the data in the `Delta` property of the first client dataset:

```
if ClientDataSet1.ChangeCount > 0 then
begin
  ClientDataSet2.Data := ClientDataSet1.Delta;
  ClientDataSet2.Open;
```

In the CdsDelta example, I've added a data module with the two ClientDataSet components and an actual source of data, a SQLDataSet mapped to InterBase's EMPLOYEE demo table. Both client datasets have the extra status calculated field, with a slightly more generic version than the code discussed earlier, because the event handler is shared between them.

TIP

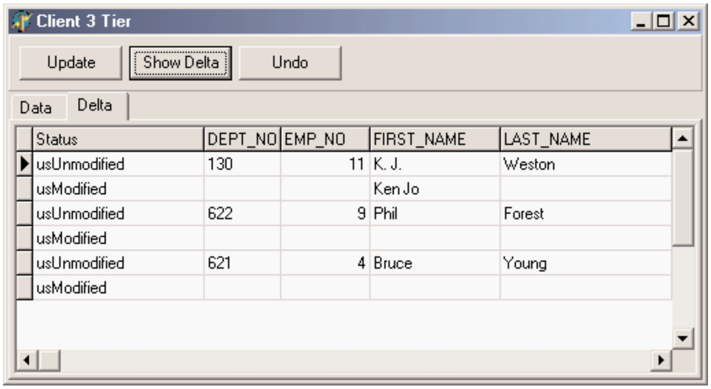
To create persistent fields for the ClientDataSet hooked to the delta (at run time), I've temporarily connected it, at design time, to the same provider of the main ClientDataSet. The structure of the delta, in fact, is the same of the dataset it refers to. After creating the persistent fields, I've removed the connection.

The form of this application has a page control with two pages, each with a DBGrid, one for the actual data and one for the delta. Some code hides or shows the second tab depending on the existence of data in the change log, as returned by the `ChangeCount` method, and updates the delta when the corresponding tab is selected. The core of the code used to handle the delta is very similar to the last code snippet above, and you can study the example source code on the CD to see more details.

You can see the change log of the CdsDelta application in Figure 14.14. Notice that the delta dataset has two entries for each modified record: the original values and the modified fields, unless this is a new or deleted record, as indicated by its status.

FIGURE 14.14:

The CdsDelta example allows you to see the temporary update requests stored in the `Delta` property of the ClientDataSet.



Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME
usUnmodified	130	11	K. J.	Weston
usModified			Ken Jo	
usUnmodified	622	9	Phil	Forest
usModified				
usUnmodified	621	4	Bruce	Young
usModified				

TIP

You can also filter the delta dataset (or any other ClientDataSet) depending on its update status, using the `StatusFilter` property. This allows you to show new, updated, and deleted records in separate grids or in a grid filtered by selecting an option in a TabControl.

Undo and *SavePoint*

Because the update data is stored in the local memory (in the delta), besides applying the updates and sending them to the application server, we can reject them, removing entries from the delta. The ClientDataSet component has a specific UndoLastChange method to accomplish this. The parameter of this method allows you to *follow* the undo operation (the name of this parameter is FollowChange). This means the client dataset will move to the record that has been restored by the undo operation.

Here is the code connected to the Undo button of the CdsDelta example:

```
procedure TForm1.ButtonUndoClick(Sender: TObject);
begin
  DmCds.cdsEmployee.UndoLastChange (True);
end;
```

An extension of the undo support is the possibility to save a sort of bookmark of the change log position (the current status) and to restore it later by undoing all successive changes. The SavePoint property can be used either to save the number of changes in the log or to reset the log to a past situation. Notice, anyway, that you can only remove records from the change log, not reinsert changes. In other words, the ChangeLog refers to a position in a log, so it can only go back to a position in which there were fewer records! This position is just a number of changes, so if you undo some changes and then do more edits, that number of changes will become meaningless.

Enabling and Disabling Logging

Keeping track of changes makes sense if you need to send the updated data back to a server database. In local applications with data stored to a MyBase file, keeping this log around can become useless and consumes memory. For this reason, you can disable logging altogether with the LogChanges property.

You can also call the MergeChangesLog method to remove all current editing from the change log. This makes sense if the dataset doesn't directly originate by a provider but was built with custom code, or in case you want to add or edit the data programmatically, without having to send it to the back-end database server.

TIP The ClientDataSet in Delphi 6 has a new property, DisableStringTrim, which allows you to keep trailing spaces in field values. In past versions, in fact, string fields were invariably trimmed, which creates trouble with some databases.

Updating the Data

Now that we have a better understanding of what goes on during local updates, we can try to make this program work by sending the local update (stored in the delta) back to the database server. To apply all the updates from a dataset at once, pass -1 to the ApplyUpdates method.

If the provider (or actually the Resolver component inside it) has trouble applying an update, it triggers the OnReconcileError event. This can take place because of a concurrent update by two different people. As we tend to use optimistic locking in client/server applications, this should be regarded as a normal situation.

The OnReconcileError event allows you to modify the Action parameter (passed by reference), which determines how the server should behave:

```
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
```

This method has three parameters: the client dataset component (in case more than one client application is interacting with the application server), the exception that caused the error (with the error message), and the kind of operation that failed (ukModify, ukInsert, or ukDelete). The return value, which you'll store in the Action parameter, can be any one of the following:

```
type TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,
  raRefresh);
```

- The raSkip value specifies that the server should skip the conflicting record, leaving it in the delta (this is the default value).
- The raAbort value tells the server to abort the entire update operation and not even try to apply the remaining changes listed in the delta.
- The raMerge value tells the server to merge the data of the client with the data on the server, applying only the modified fields of this client (and keeping the other fields modified by other clients).
- The raCorrect value tells the server to replace its data with the current client data, overriding all field changes already done by other clients.
- The raCancel value cancels the update request, removing the entry from the delta and restoring the values originally fetched from the database (thus ignoring changes done by other clients).
- The raRefresh value tells the server to dump the updates in the client delta and to replace them with the values currently on the server (thus keeping the changes done by other clients).

If you want to test a collision, you can simply launch two copies of the client application, change the same record in both clients, and then post the updates from both. We'll do this later to generate an error, but let's first see how to handle the OnReconcileError event.

This is actually a simple thing to accomplish, but only because we'll receive a little help. Since building a specific form to handle an OnReconcileError event is very common, Delphi already provides such a form in the Object Repository. Simply go to the Dialogs page and select the Reconcile Error Dialog item. This unit exports a function you can directly use to initialize and display the dialog box, as I've done in the CdsDelta example:

```
procedure TDmCds.cdsEmployeeReconcileError (DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

WARNING As the source code of the Reconcile Error Dialog unit suggests, you should use the Project Options dialog to remove this form from the list of automatically created forms (if you don't, an error will occur when you compile the project). Of course, you need to do this only if you haven't set up Delphi to skip the automatic form creation.

The `HandleReconcileError` function simply creates the form of the dialog box and shows it, as you can see in the code provided by Borland:

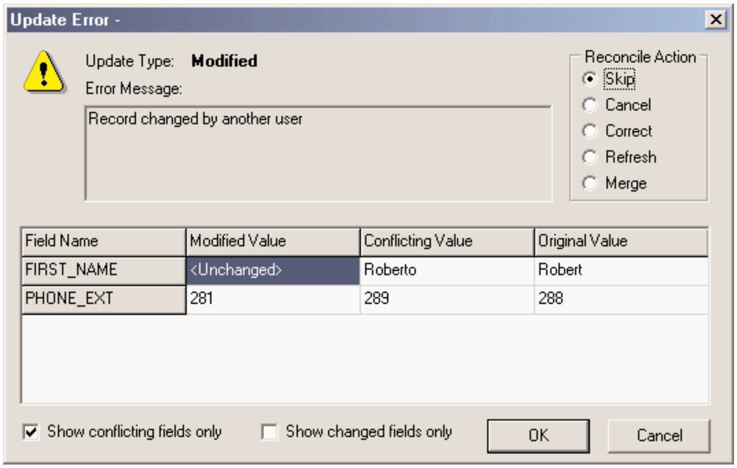
```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
  ReconcileError: EReconcileError): TReconcileAction;
var
  UpdateForm: TReconcileErrorForm;
begin
  UpdateForm := TReconcileErrorForm.CreateForm(DataSet, UpdateKind,
    ReconcileError);
  with UpdateForm do
  try
    if ShowModal = mrOK then
    begin
      Result := TReconcileAction(ActionGroup.Items.Objects[
        ActionGroup.ItemIndex]);
      if Result = raCorrect then
        SetFieldValues(DataSet);
    end
    else
      Result := raAbort;
  finally
    Free;
  end;
end;
```

The Reconc unit, which hosts the Reconcile Error dialog, contains over 350 lines of code, so we can't describe it in detail. However, you should be able to understand the source code by studying it carefully. Alternatively, you can simply use it without caring about how everything works.

The dialog box will appear in case of an error, reporting the requested change that caused the conflict and allowing the user to choose one of the possible `TReconcileAction` values. You can see an example in Figure 14.15.

FIGURE 14.15:

The Reconcile Error dialog provided by Delphi in the Object Repository and used by the CdsDelta example



Update Error -

Update Type: **Modified**

Error Message:

Record changed by another user

Reconcile Action

● Skip
○ Cancel
○ Correct
○ Refresh
○ Merge

Field Name	Modified Value	Conflicting Value	Original Value
FIRST_NAME	<Unchanged>	Roberto	Robert
PHONE_EXT	281	289	288

☑ Show conflicting fields only □ Show changed fields only

TIP When you call `ApplyUpdates`, you start a rather complex update sequence, discussed in more detail in Chapter 17 for multitier architectures. In short, the delta is sent to the provider, which fires the `OnUpdateData` event and then receives a `BeforeUpdateRecord` event for every record to update. These are two chances you have to take a look at the changes and force specific operations on the database server.

MyBase (or the Briefcase Model)

The last capability of the ClientDataSet component I want to discuss in this chapter is its support for mapping memory data to local files, building stand-alone applications. The same technique can be applied in multitier applications to use the client program even when you're not physically connected to the application server. In this case, you can save all the data you expect to need in a local file for travel with a laptop (perhaps visiting client sites). You'll use the client program to access the local version of the data, edit the data normally, and when you reconnect, apply all the updates you've performed while disconnected.

To map a ClientDataSet to a local file you only need to set its `FileName` property, which requires an absolute pathname. To build a minimal MyBase program (called MyBase1), all you need is a ClientDataSet component hooked to a file and with a few fields defined (in the `FieldDefs` property):

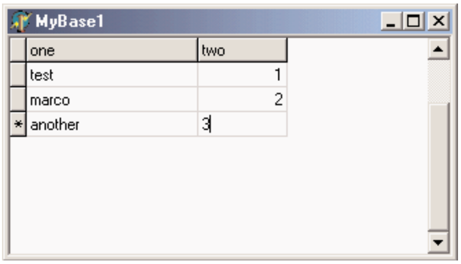
```
object ClientDataSet1: TClientDataSet
  FileName = 'C:\md6code\14\MyBase1\test'
  FieldDefs = <
    item
      Name = 'one'
      DataType = ftString
      Size = 20
    end
    item
      Name = 'two'
      DataType = ftSmallint
    end>
  StoreDefs = True
end
```

At this point you can use the Create DataSet command of the local menu of the ClientDataSet at design time, or call its `CreateDataSet` method at run time, to physically create the file for the table. As you make changes and close the application, the data will be automatically saved to the file. (You might want to disable the change log, though, to reduce the size of this data.) The dataset, in any case, also has a `SaveToFile` method and a `LoadFromFile` method you can use in your code.

MyBase1, my example program, shown in Figure 14.16, doesn't require any database server or database connection to work. It needs only your own program and the `Midas.dll` file, but you can even get rid of it by including the MidasLib unit in the project. And the program doesn't require any actual Pascal code, either.

FIGURE 14.16:

The MyBase1 example, which saves data directly to a MyBase file



	one	two
	test	1
	marco	2
*	another	3

TIP MyBase generally saves the datasets in XML format, although the internal CDS format is still available. I'll explore this format in detail when I discuss XML in Chapter 23, "XML and SOAP." For the moment, suffice to say this is a text-based format (so it is less space-efficient than the internal format), which can be manipulated programmatically but immediately makes some sense even if you try reading it.

The MyBase support in Delphi 6 also includes the possibility of extracting the XML representation of a memory dataset by using the XMLData property. In Delphi 5, you could obtain the same by saving the ClientDataSet in XML format in a memory stream.

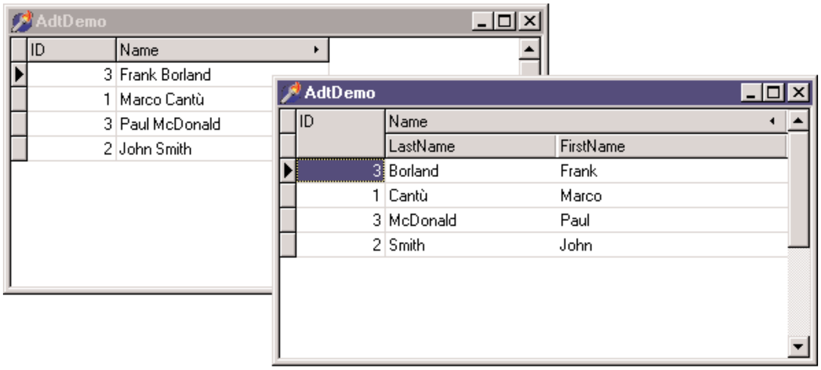
Abstract Data Types in MyBase

The ClientDataSet component supports most data types provided by Delphi, including nested data types and abstract data types, the case I want to investigate with this second MyBase example. In the FieldDefs property editor of a ClientDataSet component you can and select the ftADT value for the DataType property of one of fields. Now move to the ChildDefs property and define the child fields. This is the field definition of the AdtDemo example:

```
FieldDefs = <
  item
    Name = 'ID'
    DataType = ftInteger
  end
  item
    Name = 'Name'
    ChildDefs = <
      item
        Name = 'LastName'
        DataType = ftString
        Size = 20
      end
      item
        Name = 'FirstName'
        DataType = ftString
        Size = 20
      end>
    DataType = ftADT
    Size = 2
  end>
```

At this point, provide the FileName, create the dataset, and you are ready to compile and run the application. If you use a DBGrid to view the resulting dataset, it will allow you to expand or collapse the subfields of the ADT field, as you can see in Figure 14.17. The *condensed* value of the field is defined in the AdtDemo program by handling the OnGetText event of the ADT field:

```
procedure TForm1.ClientDataSet1NameGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := ClientDataSet1NameFirstName.AsString + ' ' +
    ClientDataSet1NameLastName.AsString;
end;
```



ID	Name
3	Frank Borland
1	Marco Cantù
3	Paul McDonald
2	John Smith

ID	Name	
	LastName	FirstName
3	Borland	Frank
1	Cantù	Marco
3	McDonald	Paul
2	Smith	John

FIGURE 14.17:

The AdtDemo example shows the support for expanding or collapsing the definition of an ADT field.

Indexing for ADT Fields

We've seen how easily you can set up an index as the user selects the title of a DBGrid. In ADT fields, the situation becomes a little more complex. The AdtDemo program, in fact, uses the `FullName` property of the field (not the `FieldName` property) because of the ADT definition. For the `LastName` child field, in fact, the index should be based on `Name.LastName`, not simply on `LastName`. Also, the ADT field cannot itself be indexed, so if it is selected, the program uses as index the `LastName` subfield. Here is the code:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  if Column.Field.FullName = 'Name' then
    ClientDataSet1.IndexFieldNames := 'Name.LastName'
  else
    ClientDataSet1.IndexFieldNames := Column.Field.FullName;
end;
```

What's Next?

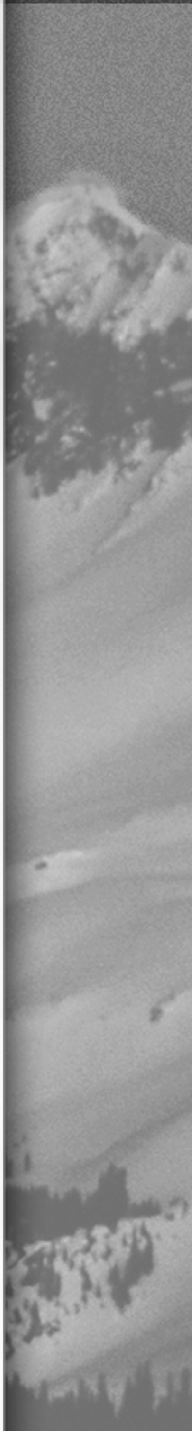
This chapter has presented a somewhat detailed introduction to client/server programming with Delphi. We saw what the key issues are and delved a little into some interesting areas of client/server programming. After a general introduction, I discussed the use of the dbExpress database library Borland is introducing in Delphi 6 and of the ClientDataSet component and MyBase technology.

There is certainly more we can say about client/server programming in Delphi, and in the next chapter I'll discuss some real-world examples, after introducing InterBase and the IBX components. Chapter 16 will then focus on Microsoft's ADO database engine.

CHAPTER 15

InterBase and IBX

- Getting started with InterBase 6
- Server-side programming: views, stored procedures, and triggers
- Using InterBase Express
- Pieces for a real-world example



Client/server programming requires two sides: a client application that you probably want to build with Delphi, and a relational database management system (RDBMS), usually a "SQL server." In this chapter, I focus on one specific SQL server, InterBase. There are many reasons for this choice. InterBase is the SQL server developed by Borland; it is an open source project and can be obtained for free; and it has traditionally been bound with Delphi, which has specific dataset components for it.

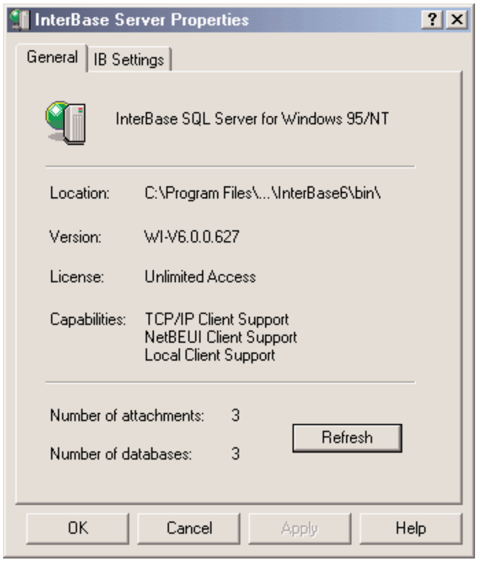
For all of these reasons, InterBase should be a good choice for your Delphi client/server development, although there are many other equally powerful alternatives. I'll discuss InterBase from the Delphi perspective, without delving in to its internal architecture. A lot of the information presented also applies to other SQL servers, so even if you've decided not to use InterBase, you might still find it valuable.

Getting Started with InterBase 6

After installing InterBase 6, you'll be able to activate the server from the Windows Start menu, but if you plan on using it frequently, you should install it as a Windows service (of course, only if you have Windows NT/2000, as Windows 9*x*/Me doesn't have support for services). When the server is active, you'll see a corresponding icon in the Tray Icon area of the Windows Taskbar (unless you start it as a service). The menu connected with this icon allows you to see status information (see Figure 15.1) and do some very limited configuration.

FIGURE 15.1:

The status information displayed by InterBase when you double-click its tray icon


InterBase Server Properties

InterBase SQL Server for Windows 95/NT

Location: C:\Program Files\...\InterBase6\bin\

Version: WI-V6.0.0.627

License: Unlimited Access

Capabilities: TCP/IP Client Support
NetBEUI Client Support
Local Client Support

Number of attachments: 3

Number of databases: 3

Inside InterBase

Even though it has a limited market share, InterBase is a very powerful RDBMS. In this section I'll introduce the key technical features of InterBase, without getting into too much detail. This is a book on Delphi programming, in fact. Unfortunately, there is currently very little published about InterBase, although there are some ongoing efforts for an InterBase book and there is a wealth of information in the documentation accompanying the product and on a few Web sites devoted to the product.

InterBase was built from the beginning with a very modern and robust architecture. Its original author, Jim Starkey, invented an architecture for handling concurrency and transactions without imposing physical locks on portions of the tables, something other well-known database servers can hardly do even today. InterBase architecture is called Multi-Generational Architecture (MGA), and it handles concurrent access to the same data by multiple users, who can modify records without affecting what other concurrent users see in the database.

This approach naturally maps to the *Repeatable Read* transaction isolation mode, in which a user within a transaction keeps seeing the same data, regardless of changes done and committed by other users. Technically, the server handles this by maintaining a different version of each accessed record for each open transaction. Even if this approach (also called *versioning*) can lead to larger memory consumption, it avoids almost any physical lock on the tables and makes the system much more robust in case of a crash. Also, MGA pushes toward a very clear programming model—Repeatable Read—which other well-known SQL servers don't even support without losing most of their performance.

If Multi-Generational Architecture is at the heart of InterBase, the server has many other technical advantages:

- A limited footprint, which makes InterBase the ideal candidate for running directly on client computers, including portables. The disk space required by InterBase for a minimal installation is well below 10 MB, and its memory requirements are also incredibly limited.
- Good performance on large amounts of data.
- Availability on many different platforms (including 32-bit Windows, Solaris, and Linux), with totally compatible versions, which makes the server scalable from very small to huge systems without notable differences.
- A very good track record, as InterBase has been in use for 15 years with very few problems.
- A language very close to the SQL standard.

- Advanced programming capabilities, with positional triggers, selectable stored procedures, updateable views, exceptions, events, generators, and more.
- Simple installation and management, with limited administration headaches.

A Short History of InterBase

Jim Starkey wrote InterBase for his own Groton Database Systems company (hence the `.gds` extension still in use for InterBase files). The company was later bought by Ashton-Tate, which was then acquired by Borland. Borland handled InterBase directly for a while, then created an InterBase subsidiary, which was later re-absorbed into the parent company.

Starting with Delphi 1, an evaluation copy of InterBase has been distributed along with the development tool, spreading the database server among developers. Although it doesn't have a large piece of the RDBMS market, which is dominated by a handful of players, InterBase has been chosen by a few very relevant organizations, from Ericsson to the U.S. Department of Defense, from stock exchanges to home banking systems.

More recent events include the announcement of InterBase 6 as an open source database (December 1999), the effective release of source code to the community (July 2000), and the release of the officially certified version of InterBase 6 by Borland (March 2001).

In between these events, there were announcements of the spin-off of a separate company to run the consulting and support business on top of the open source database. Contacts with a group of former InterBase developers and managers (who had left Borland) didn't lead to an agreement, but the group decided to go ahead even without Borland's help and formed IBPhoenix (`www.ibphoenix.com`) with the plan of supporting InterBase users.

At the same time, independent groups of InterBase experts formed the InterBase Developer Initiative (IBDI; `www.interbase2000.org`) and started the Firebird open source project to further extend InterBase. For this reason, SourceForge currently hosts two different versions of the project, InterBase itself run by Borland and the Firebird project run by this independent group. You see that the picture is rather complex, but this certainly isn't a problem for InterBase, as there are currently many organizations pushing it, along with Borland.

IBConsole

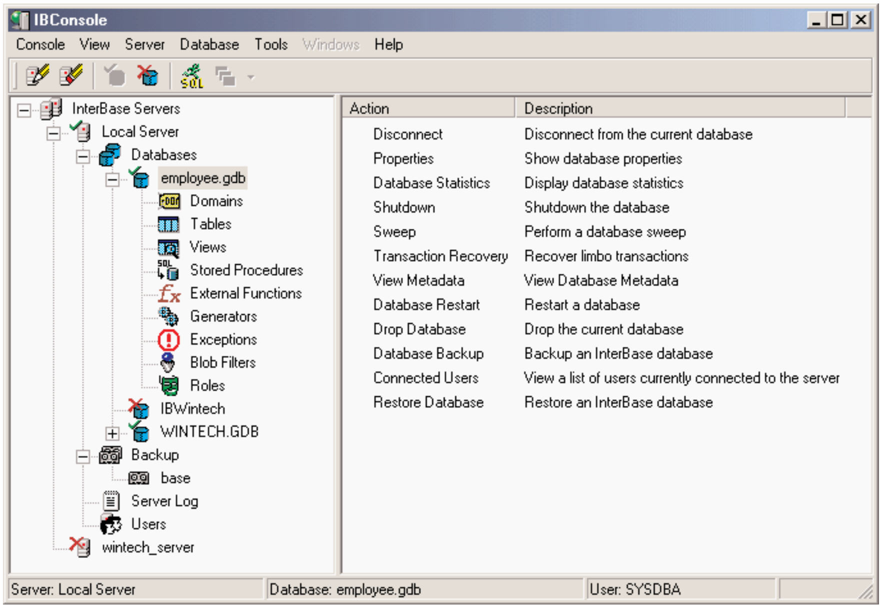
In past versions of InterBase, there were two main tools you could use to interact directly with the program: the Server Manager application, which could be used to administer both a local and a remote server; and Windows Interactive SQL (WISQL). Version 6 includes a much more powerful front-end application, called IBConsole. This is a full-fledged Windows program (built with Delphi) that allows you to administer, configure, test, and query an InterBase server, whether local or remote.

IBConsole is a simple and complete system for managing InterBase servers and their databases. You can use it to look into the details of the database structure, modify it, query the data (which can be useful to develop the queries you want to embed in your program), back up and restore the database, and perform all the other administrative tasks.

As you can see in Figure 15.2, IBConsole allows you to manage multiple servers and databases, all listed in a single, handy configuration tree. You can ask for general information about the database and list its entities (tables, domains, stored procedures, triggers, and everything else), accessing the details of each. You can also create new databases and configure them, back up the files, update the definitions, check what's going on and who is currently connected, and so on.

FIGURE 15.2:

IBConsole allows you to manage, from a single computer, InterBase databases hosted by multiple servers.



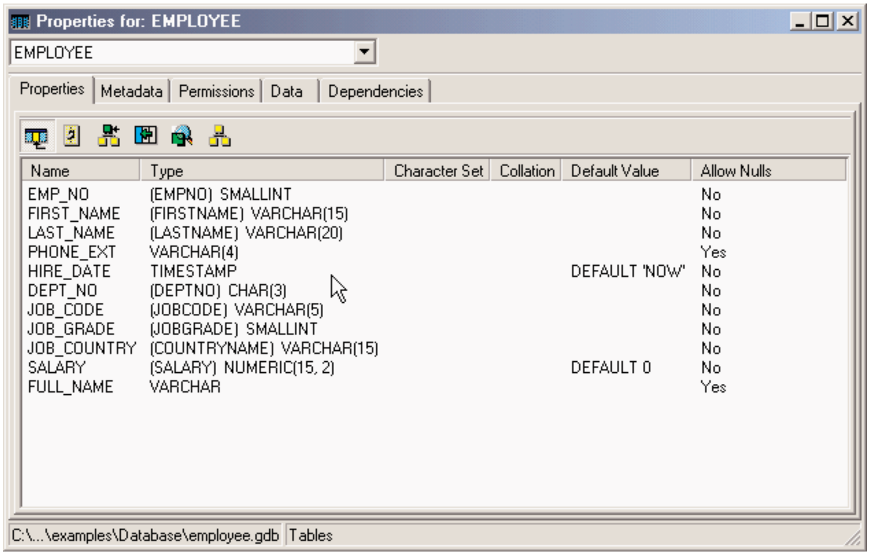
Action	Description
Disconnect	Disconnect from the current database
Properties	Show database properties
Database Statistics	Display database statistics
Shutdown	Shutdown the database
Sweep	Perform a database sweep
Transaction Recovery	Recover limbo transactions
View Metadata	View Database Metadata
Database Restart	Restart a database
Drop Database	Drop the current database
Database Backup	Backup an InterBase database
Connected Users	View a list of users currently connected to the server
Restore Database	Restore an InterBase database

Server: Local Server | Database: employee.gdb | User: SYSDBA

The IBConsole application allows you to open multiple windows to look at detailed information, such as the tables window depicted in Figure 15.3. In this window, you can see lists of the key properties of each table (columns, triggers, constraints, and indexes), see the raw metadata (the SQL definition of the table), access permissions, have a look at the actual data, modify it, and study the dependencies of the table. Similar windows are available for each of the other entities you can define in a database.

FIGURE 15.3:

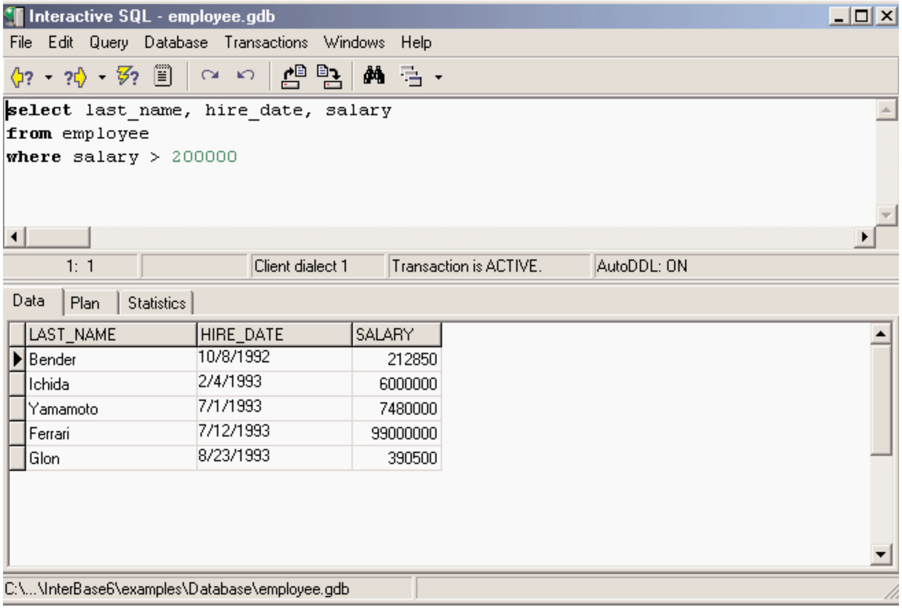
IBConsole can open separate windows to show you the details of each entity—in this case, a table.



Name	Type	Character Set	Collation	Default Value	Allow Nulls
EMP_NO	(EMPNO) SMALLINT				No
FIRST_NAME	(FIRSTNAME) VARCHAR(15)				No
LAST_NAME	(LASTNAME) VARCHAR(20)				No
PHONE_EXT	VARCHAR(4)				Yes
HIRE_DATE	TIMESTAMP			DEFAULT 'NOW'	No
DEPT_NO	(DEPTNO) CHAR(3)				No
JOB_CODE	(JOBCODE) VARCHAR(5)				No
JOB_GRADE	(JOBGRADE) SMALLINT				No
JOB_COUNTRY	(COUNTRYNAME) VARCHAR(15)				No
SALARY	(SALARY) NUMERIC(15, 2)			DEFAULT 0	No
FULL_NAME	VARCHAR				Yes

Finally, IBConsole embeds an improved version of the original Windows Interactive SQL application (see Figure 15.4). You can directly type a SQL statement in the upper portion of the window (without any actual help from the tool, unfortunately) and then execute the SQL query. As a result, you'll see the data, but also the access *plan* used by the database (which an expert can use to determine the efficiency of the query) and some statistics on the actual operation performed by the server.

This is really a minimal description of IBConsole, which is a rather powerful tool and the only one included by Borland with the server besides command-line tools. IBConsole is probably not the most complete tool in its category, though. Quite a few third-party InterBase management applications are more powerful, although they are not all very stable or user-friendly. Some InterBase tools are shareware programs, while others are totally free. Two examples, out of many, are InterBase Workbench (`www.interbaseworkbench.com`) and IB_WISQL (done with and part of InterBase Objects, `www.ibobjects.com`).


Interactive SQL - employee.gdb
File Edit Query Database Transactions Windows Help
Data
Plan
Statistics

```
select last_name, hire_date, salary
from employee
where salary > 200000
```

1: 1 | Client dialect 1 | Transaction is ACTIVE. | AutoDDL: ON

LAST_NAME	HIRE_DATE	SALARY
Bender	10/8/1992	212850
Ichida	2/4/1993	6000000
Yamamoto	7/1/1993	7480000
Ferrari	7/12/1993	99000000
Glon	8/23/1993	390500

C:\...\InterBase6\examples\Database\employee.gdb

FIGURE 15.4: The Interactive SQL window of IBConsole allows you to try out in advance the queries you plan to include in your Delphi programs.

TIP To find the latest third-party InterBase tools, have a look at `www.interbase2000.org/tools`, which hosts an up-to-date list.

Server-Side Programming

At the beginning of the previous chapter, I underlined the fact that one of the objectives of client/server programming—and one of its problems—is the division of the workload between the computers involved. When you activate SQL statements from the client, the burden falls on the server to do most of the work. However, you should try to use `select` statements that return a large result set, to avoid jamming the network.

Besides accepting DDL and DML requests, most RDBMS servers allow you to create routines directly on the server using the standard SQL commands plus their own server-specific extensions (which are generally not portable). These routines typically come in two forms, stored procedures and triggers.

Stored Procedures

Stored procedures are like the global functions of a Delphi unit and must be explicitly called by the client side. Stored procedures are generally used to define routines for data maintenance, to group sequences of operations you need in different circumstances, or to hold complex `select` statements.

Like Pascal procedures, stored procedures can have one or more typed parameters. Unlike Pascal procedures, they can have more than one return value. As an alternative to returning a value, a stored procedure can also return a result set, the result of an internal `select` statement or a custom fabricated one.

The following is a stored procedure written for InterBase; it receives a date in input and computes the highest salary among the employees hired on that date:

```
create procedure maxsaloftheday(ofday date)
returns (maxsal decimal(8,2)) as
begin
  select max(salary)
  from employee
  where hiredate = :ofday
  into :maxsal;
end
```

Notice the use of the `into` clause, which tells the server to store the result of the `select` statement in the `maxsal` return value. To modify or delete a stored procedure, you can later use the `alter procedure` and `drop procedure` commands.

Looking at this stored procedure, you might wonder what its advantage is compared to the execution of a similar query activated from the client. The difference between the two approaches is not in the result you obtain but in its speed. A stored procedure is compiled on the server in an intermediate and faster notation when it is created, and the server determines at that time the strategy it will use to access the data. By contrast, a query is compiled every time the request is sent to the server. For this reason, a stored procedure can replace a very complex query, provided it doesn't change too often!

From Delphi you can activate a stored procedure returning a result set by using either a Query or a StoredProc component. With a Query, you can use the following SQL code:

```
select *
from MaxSalOfTheDay ('01/01/1990')
```

Triggers (and Generators)

Triggers behave more or less like Delphi events and are automatically activated when a given *event* occurs. Triggers can have specific code or call stored procedures; in both cases, the execution is done completely on the server. Triggers are used to keep data consistent, checking new data in more complex ways than a check constraint allows, and to automate the side effects of some input operations (such as creating a log of previous salary changes when the current salary is modified).

Triggers can be fired by the three basic data update operations: `insert`, `update`, and `delete`. When you create a trigger, you indicate whether it should fire before or after one of these three actions.

As an example of a trigger, we can use a generator to create a unique index in a table. Many tables use a unique index as primary key. InterBase doesn't have an AutoInc field, unlike Paradox and other local databases. Because multiple clients cannot generate unique identifiers, we can rely on the server to do this. Almost all SQL servers offer a counter you can call to ask for a new ID, which you should later use for the table. InterBase calls these automatic counters *generators*, while Oracle calls them *sequences*. Here is the sample InterBase code:

```
create generator cust_no_gen;
...
gen_id (cust_no_gen, 1);
```

The `gen_id` function then extracts the new unique value of the generator passed as first parameter, with the second parameter indicating how much to increase (in this case, by one).

At this point you can add a trigger to a table, an automatic handler for one of the table's events. A trigger is similar to the event handler of the Table component, but you write it in SQL and execute it on the server, not on the client. Here is an example:

```
create trigger set_cust_no for customers
before insert position 0 as
begin
  new.cust_no = gen_id (cust_no_gen, 1);
end
```

This trigger is defined for the Customer table and is activated each time a new record is inserted. The `new` symbol indicates the new record we are inserting. The `position` option indicates the order of execution of multiple triggers connected to the same event. Triggers with the lowest values will be executed first.

Inside a trigger, you can write DML statements that also update other tables, but watch out for updates that end up reactivating the trigger, creating an endless recursion. You can later modify or disable a trigger by calling the `alter trigger` statement or `drop trigger`.

TIP Triggers fire automatically for specified events. If you have to make many changes in the database using batch operations, the presence of a trigger might slow down the process. If the input data has already been checked for consistency, you can temporarily deactivate the trigger. These batch operations are often coded in stored procedures, but stored procedures generally cannot issue DDL statements, like those required for deactivating and reactivating the trigger. In this situation, you can define a view based on a simple `select * from table` command, thus creating an alias for the table. Then you can let the stored procedure do the batch processing on the table and apply the trigger to the view (which should also be used by the client program).

Using InterBase Express

The examples built in the last chapter either still used the BDE or were done with the new dbExpress database engine. Using this server-independent engine could allow you to switch the database server used by your application, although in practice this is often far from simple. You might decide that an application you are building will invariably use a given database server, possibly the internal server of the company you are working for. In this case, you can decide to skip any database engine or library as well and write programs that are tied directly to the API of the specific database server, which will make your program intrinsically nonportable to other SQL servers.

Of course, you won't generally use similar APIs directly, but rather base your development on some native or third-party dataset components, which wrap these APIs and naturally fit into Delphi and the architecture of its class library. An example of such a family of components is InterBase Express (IBX). Applications built using these components should work better and faster (even if only marginally), giving you more control over the specific features of the server. For example, IBX provides you a set of administrative components specifically built for InterBase 6.

NOTE I'll examine the IBX components because they are tied to InterBase (the database server discussed in this chapter) and because that set is the only one available in the standard Delphi installation. Other similar sets of components (for InterBase, Oracle, and other database servers) are equally powerful and well-regarded in the Delphi programmers' community. A good example (and an alternative to IBX) is InterBase Objects, `www.ibobjects.com`.

IBX Dataset Components

The IBX components include custom dataset components and a few others. The dataset components inherit from the base `TDataSet` class, can use all the common Delphi data-aware controls, provide a field editor and all the usual design-time features, and can be used in the Data Module Designer, but they don't require the BDE.

You can actually choose among multiple dataset components. Three datasets of IBX have a role and a set of properties similar to their BDE counterparts:

- IBTable resembles the Table component and allows you to access a single table or view.
- IBQuery resembles the Query component and allows you to execute a SQL query, returning a result set. The IBQuery component can be used together with the IBUpdateSQL component to obtain a live (or editable) dataset.
- IBStoredProc resembles the StoredProc component and allows you to execute a stored procedure.

For new applications, you should generally use the IBDataSet component, which allows you to work with a live result set obtained by executing a `select` query. It basically merges IBQuery with IBUpdateSQL in a single component. The three components above, in fact, are provided mainly for compatibility with Delphi BDE applications.

Many other components in InterBase Express don't belong to the dataset category, but are still used in applications that need to access to a database:

- IBDatabase mimics the BDE Database component and is used to set up the database connection. The BDE also uses the specific Session component to perform some global tasks done by the IBDatabase component.
- IBTransaction allows complete control over transactions. It is important in InterBase to use transactions explicitly and isolate each transaction properly, using the Snapshot isolation level for reports and the Read Committed level for interactive forms. Each dataset explicitly refers to a given transaction, so you can have multiple concurrent transactions against the same database, choosing which datasets take part in which transaction.
- IBSQL lets you execute SQL statements that don't return a dataset (for example, DDL requests, or update and delete statements) without the overhead of a dataset component.
- IBDatabaseInfo is used for querying the database structure and status.
- IBSQLMonitor is used for debugging the system, since the SQL Monitor debugger provided by Delphi is a BDE-specific tool.
- IBEvents receives events posted by the server.

This group of components provides greater control over the database server than you can have with the BDE. For example, having a specific transaction component allows you to manage multiple concurrent transactions over one or multiple databases, as well as a single transaction spanning multiple databases. The IBDatabase component allows you to create databases, test the connection, and generally access system data, something the Database and Session BDE components don't fully provide.

TIP A feature of the IBX datasets that is new in Delphi 6 is the ability to set up the automatic behavior of a generator as a sort of auto-incremental field. This is accomplished by setting the `GeneratorField` property using its specific property editor. An example of this is discussed later in this chapter in the section "Generators and IDs."

IBX Administrative Components

A new page of Delphi 6 Component palette, InterBase Admin, hosts InterBase 6 administrative components. Although your aim is probably not to build a full InterBase console application, including some administrative features (such as backup handling or user monitoring) can make sense in applications meant for power users.

Most of these components have self-explanatory names. They are IBConfigService, IBBackupService, IBRestoreService, IBValidationService, IBStatisticalService, IBLogService, IBSecurityService, IBServerProperties, IBInstall, and IBUninstall. I won't build any advanced examples of the use of these components, as they are more focused towards the development of server management applications than that of client programs. I'll only embed a couple of them in a simple example, later in this chapter.

From BDE to IBX

To demonstrate how simple it can be to move from the use of the BDE to the use of IBX, I've built a trivial application, using the Database Form Wizard (which is strictly bound to the BDE). The application, on the companion CD, is called IbEmp and shows only a few fields of the *usual* Employee table of the corresponding InterBase demo database.

All of the features of the IbEmp example are summarized by the properties of its Query component:

```
object Query1: TQuery
  DatabaseName = 'IBLocal'
  RequestLive = True
  SQL.Strings = (
    'SELECT * '
    'FROM EMPLOYEE')
end
```

I could have extended the structure of this example generated by Delphi, adding a Database component to handle the connection, but I decided this was useless, as my intention was only to port the example to the use of the IBX components. The interesting example, in fact, is IbEmp2, which I started by copying all of the source code files of the version generated by the wizard. (The previous example is available on the companion CD just so you can try this type of porting, as the example by itself is not particularly interesting.)

After replacing the Query component with an IBQuery, I had to add two more components: IBTransaction and IBDatabase. Any IBX application requires at least an instance of each of these two components. You cannot set database connections in a dataset (as you can do with a plain Query), and at least a transaction object is required even to read the result of a query.

Here are the key properties of these components in the IbEmp2 example:

```
object IBTransaction1: TIBTransaction
  Active = False
  DefaultDatabase = IBDatabase1
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  SQL.Strings = (
    'SELECT * FROM EMPLOYEE')
end
object IBDatabase1: TIBDatabase
  DatabaseName = 'C:\Program Files\InterBase ' +
    'Corp\InterBase6\examples\Database\employee.gdb'
  Params.Strings = (
    'user_name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  IdleTimer = 0
  SQLDialect = 1
  TraceFlags = []
end
```

The changes don't take too much time to perform, and if you are accessing the same database table as in the BDE-based program, you won't need to change the data-aware components at all, but only hook the DataSource component to IBQuery1. Because I'm not using the BDE, I had to type in the pathname of the InterBase database. However, not everyone in the world has the Program Files folder, which depends on the local version of Windows, and of course the InterBase sample data files could have been installed in any other location of the disk. We'll try to solve these problems in the next example.

WARNING Notice that I've embedded the password in the code, a very naïve approach to security. Not only can anyone run the program, but someone could even extract the password by looking at the hexadecimal code of the executable file. I used this approach so I wouldn't need to keep typing in my password while testing a program, but in a real application you should require your users to do so if they care about the security of their data.

Building a Live Query

The IbEmp2 example has a query that doesn't allow editing. To activate editing, you need to use an IBTable component or add to the query an IBUpdateSQL component, even if the query is very simple. Usually the BDE does the behind-the-scenes work that lets you edit the result set of a simple query, but we are not using the BDE now.

The relationship between the IBQuery and IBUpdateSQL components is the same as between the Query and UpdateSQL components. To highlight this, I've taken the main form of the UpdateSql example discussed in the last chapter and ported it to the InterBase Express components, building the UpdSql2 example. I've simply copied the two components from the original example, pasted them into an editor, changed the type of the object, and copied the resulting text into a new form. The properties are so similar that I had only to ignore a couple of missing ones (the `DatabaseName` and the `UpdateMode` properties).

At this point, I simply added an IBDatabase and an IBTransaction component, a data source and a grid, and my program was up and running. The key element of these components, in fact, is their SQL code, which is attached to the `SQL` property of the query and the `ModifySQL`, `DeleteSQL`, and `InsertSQL` properties of the update component.

However, this time I've made the reference to the database a little more flexible. Instead of typing in the database name at design time, I've extracted the `InterBase` folder from the Windows Registry (where Borland saves it while installing the programs). This is the code executed when the program starts:

```
uses
  Registry;

procedure TForm1.FormCreate(Sender: TObject);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    Reg.OpenKey('\Software\Borland\InterBase\CurrentVersion', False);
    IBDatabase1.DatabaseName := Reg.ReadString('RootDirectory') +
      'examples\database\employee.gdb';
    finally
      Reg.CloseKey;
      Reg.Free;
    end;
    EmpDS.DataSet.Open;
  end;
```

The source code actually contains alternate code for using the database installed in the `Data` subfolder of the `Borland Shared` folder, used for Delphi sample databases. Notice also that InterBase 6 places the sample databases in a different subfolder than InterBase 5 did.

NOTE For more information about the Windows Registry and INI files, see the related sidebar in Chapter 10, "The Architecture of Delphi Applications."

The new feature of this example, compared to the last version, is the presence of a transaction component. As I've already said, the InterBase Express components make the use of a transaction component compulsory, explicitly following a requirement of InterBase. Simply adding a couple of buttons to the form to commit or roll back the transaction would be enough, because a transaction starts automatically as you edit any dataset attached to it.

I've also improved the program a little by adding an ActionList component to it. This includes all the standard database actions and adds two custom actions for transaction support, Commit and Rollback. Both actions are enabled when the transaction is active:

```
procedure TForm1.ActionUpdateTransactions(Sender: TObject);
begin
  acCommit.Enabled := IBTransaction1.InTransaction;
  acRollback.Enabled := acCommit.Enabled;
end;
```

When executed, they perform the main operation but also need to reopen the dataset in a new transaction (which can also be done by "retaining" the transaction context). Actually, `CommitRetaining` doesn't reopen a new transaction, but it allows the current transaction to remain open. This way, you can keep using your datasets, which won't be refreshed (so you won't see edits already committed by other users) but will keep showing the data you've modified. This is the code:

```
procedure TForm1.acCommitExecute(Sender: TObject);
begin
  IBTransaction1.CommitRetaining;
end;

procedure TForm1.acRollbackExecute(Sender: TObject);
begin
  IBTransaction1.Rollback;
```

```
    // reopen the dataset in a new transaction
    IBTransaction1.StartTransaction;
    EmpDS.DataSet.Open;
  end;
```

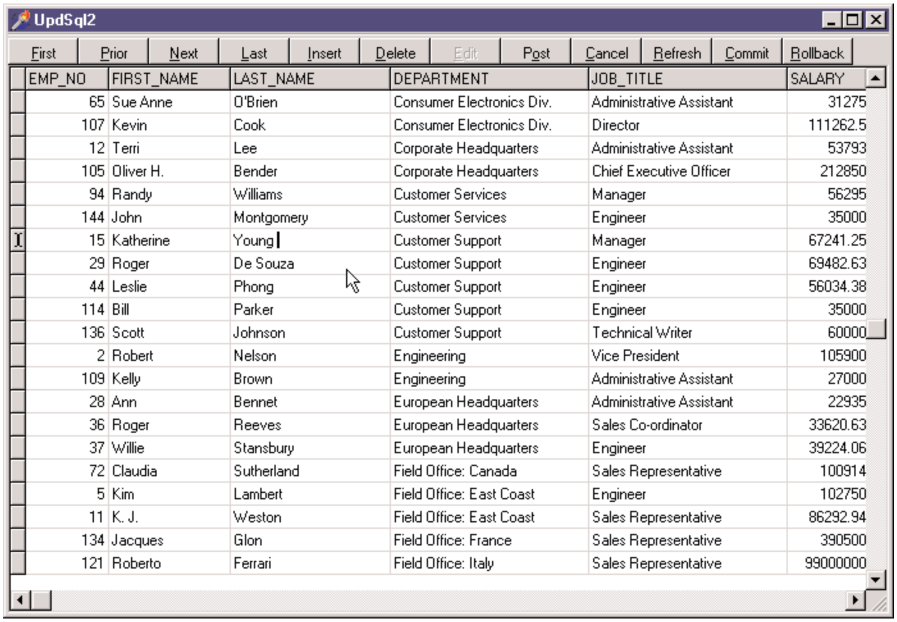
WARNING Be aware that InterBase closes any opened cursors when a transaction ends, which means you have to reopen them and refetch the data even if you haven't made any changes. When committing data, instead, you can ask InterBase to retain the "transaction context"—not to close open datasets—by issuing a `CommitRetaining` command, as mentioned before. The reason for this behavior of InterBase depends on the fact that a transaction corresponds to a snapshot of the data. Once a transaction is finished, you are supposed to read the data again to refetch records that may have been modified by other users. Version 6.0 of InterBase includes also a `RollbackRetaining` command, which I've decided not to use, because in a rollback operation, the program should refresh the dataset data to show the original values on screen, not the updates you've discarded.

The last operation refers to a generic dataset and not a specific one because I'm going to add a second alternate dataset to the program. The actions are connected to a text-only toolbar, as you can see in Figure 15.5. The program opens the dataset at startup and automatically closes the current transaction on exit, after asking the user what to do, with the following `OnClose` event handler:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  nCode: Word;
begin
  if IBTransaction1.InTransaction then
  begin
    nCode := MessageDlg ('Commit Transaction? (No to rollback)',
      mtConfirmation, mbYesNoCancel, 0);
    case nCode of
      mrYes: IBTransaction1.Commit;
      mrNo: IBTransaction1.Rollback;
      mrCancel: Action := caNone; // don't close
    end;
  end;
end;
```

An alternative to using the IBQuery and IBUpdateSQL components is to use the IBDataSet component, which combines the two. An InterBase dataset, in fact, is a live query with a complete set of SQL statements for all the main operations. The differences between using the two components and the single one are minimal. Using IBQuery and IBUpdateSQL is probably better when porting an existing application based on the two equivalent BDE components, even if porting the program directly to the IBDataSet component doesn't really require a lot of extra work.

FIGURE 15.5: The output of the UpdSql2 example



EMP_NO	FIRST_NAME	LAST_NAME	DEPARTMENT	JOB_TITLE	SALARY
65	Sue Anne	O'Brien	Consumer Electronics Div.	Administrative Assistant	31275
107	Kevin	Cook	Consumer Electronics Div.	Director	111262.5
12	Terri	Lee	Corporate Headquarters	Administrative Assistant	53793
105	Oliver H.	Bender	Corporate Headquarters	Chief Executive Officer	212850
94	Randy	Williams	Customer Services	Manager	56295
144	John	Montgomery	Customer Services	Engineer	35000
15	Katherine	Young	Customer Support	Manager	67241.25
29	Roger	De Souza	Customer Support	Engineer	69482.63
44	Leslie	Phong	Customer Support	Engineer	56034.38
114	Bill	Parker	Customer Support	Engineer	35000
136	Scott	Johnson	Customer Support	Technical Writer	60000
2	Robert	Nelson	Engineering	Vice President	105900
109	Kelly	Brown	Engineering	Administrative Assistant	27000
28	Ann	Bennet	European Headquarters	Administrative Assistant	22935
36	Roger	Reeves	European Headquarters	Sales Co-ordinator	33620.63
37	Willie	Stansbury	European Headquarters	Engineer	39224.06
72	Claudia	Sutherland	Field Office: Canada	Sales Representative	100914
5	Kim	Lambert	Field Office: East Coast	Engineer	102750
11	K. J.	Weston	Field Office: East Coast	Sales Representative	86292.94
134	Jacques	Glon	Field Office: France	Sales Representative	390500
121	Roberto	Ferrari	Field Office: Italy	Sales Representative	99000000

In the UpdSql2 example, I've provided both alternatives, so that you can test the differences yourself. Here is part of the DFM description of the dataset component:

```
object IBDataSet1: TIBDataSet
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  BufferChunks = 32
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '  (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, ' +
    '  JOB_COUNTRY)'
    'values'
```

```
      '  (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, ' +
      '  :JOB_GRADE, :JOB_COUNTRY)')
    SelectSQL.Strings = (...)
    UpdateRecordTypes = [cusUnmodified, cusModified, cusInserted]
    ModifySQL.Strings = (...)
  end
```

If you connect the `IBQuery1` or the `IBDataSet1` components to the data source and run the program, you'll see that the behavior is identical. Not only do the components have a similar effect; the available properties and events are also very similar.

Monitoring InterBase Express

SQL Monitor works by using a hook into the BDE architecture. For this reason, you cannot use it with applications based on the InterBase Express components. Instead, you can simply embed in your application a copy of the IBSQLMonitor component and produce a custom log.

You can even write a more generic monitoring application, as I've done in the IbxMon example. I've placed in its form a monitoring component and a RichEdit control, and written the following handler for the `OnSQL` event:

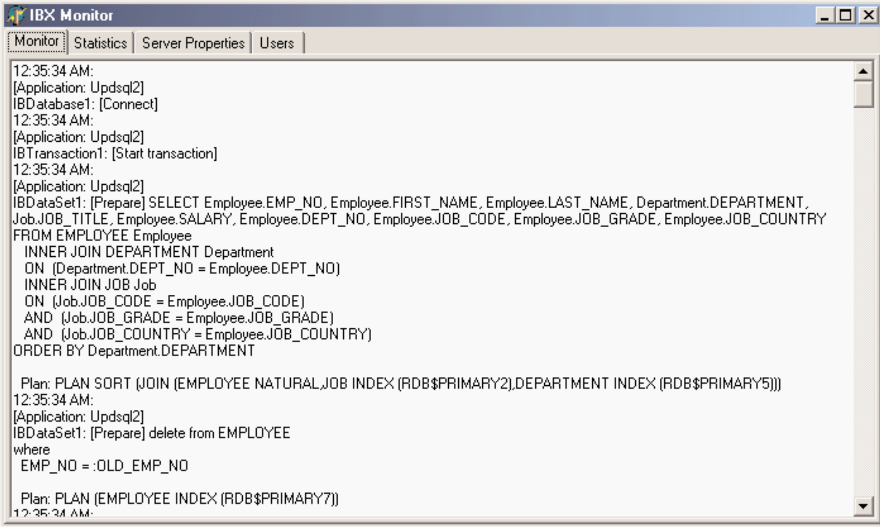
```
procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
  if Assigned (RichEdit1) then
    RichEdit1.Lines.Add (TimeToStr (Now) + ': ' + EventText);
end;
```

The `if Assigned` test can be useful when receiving a message during shutdown, and it is required when you add this code directly inside the application you are monitoring.

To receive the messages from other applications (or from the current one), you have to turn on the tracing options of the IBDatabase component. In the UpdSql2 example (discussed earlier, in the section "Building a Live Query"), I turned them all on:

```
object IBDatabase1: TIBDatabase
  ...
  TraceFlags = [tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
                tfConnect, tfTransact, tfBlob, tfService, tfMisc]
```

If you run the two examples at the same time, the output of the IbxMon program will list the details about the UpdSql2 program's interaction with InterBase, as you can see in Figure 15.6.



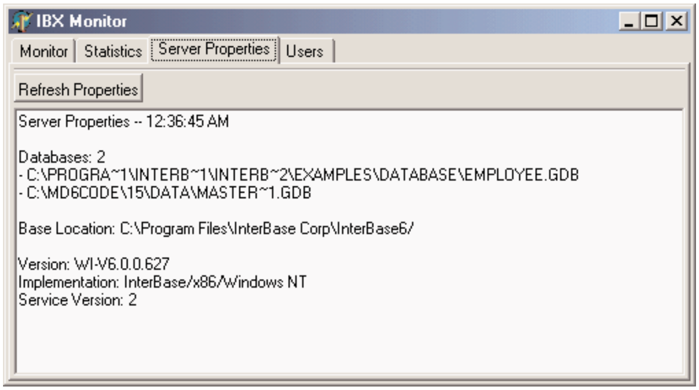
```
12:35:34 AM:
[Application: Updsql2]
IBDatabase1: [Connect]
12:35:34 AM:
[Application: Updsql2]
IBTransaction1: [Start transaction]
12:35:34 AM:
[Application: Updsql2]
IBDataSet1: [Prepare] SELECT Employee.EMP_NO, Employee.FIRST_NAME, Employee.LAST_NAME, Department.DEPARTMENT,
Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY
FROM EMPLOYEE Employee
   INNER JOIN DEPARTMENT Department
   ON  (Department.DEPT_NO = Employee.DEPT_NO)
   INNER JOIN JOB Job
   ON  (Job.JOB_CODE = Employee.JOB_CODE)
   AND  (Job.JOB_GRADE = Employee.JOB_GRADE)
   AND  (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)
ORDER BY Department.DEPARTMENT

  Plan: PLAN SORT (JOIN (EMPLOYEE NATURAL,JOB INDEX (RDB$PRIMARY2),DEPARTMENT INDEX (RDB$PRIMARY5)))
12:35:34 AM:
[Application: Updsql2]
IBDataSet1: [Prepare] delete from EMPLOYEE
where
  EMP_NO = :OLD_EMP_NO

  Plan: PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
12:35:34 AM:
```

FIGURE 15.6:

The output of the IbxMon example, based on the IBMonitor component

Getting More System Data

The IbxMon example doesn't only monitor the InterBase connection, but it allows you also to query some settings to the server using the various tabs of its page control. The example embeds a few IBX administrative components, showing server statistics, a few server properties, and all connected users. You can see an example of server properties in Figure 15.7 and the code for extracting the users in the following code fragment.



```
Server Properties -- 12:36:45 AM

Databases: 2
- C:\PROGRA~1\INTERB~1\INTERB~2\EXAMPLES\DATABASE\EMPLOYEE.GDB
- C:\MD6CODE\15\DATA\MASTER~1.GDB

Base Location: C:\Program Files\InterBase Corp\InterBase6/

Version: WI-V6.0.0.627
Implementation: InterBase/x86/Windows NT
Service Version: 2
```

FIGURE 15.7:

The assorted server information displayed by the IbxMon application

```
// grab the users data
IBSecurityService1.DisplayUsers;
// display the name of each user
for i := 0 to IBSecurityService1.UserInfoCount - 1 do
  with IBSecurityService1.UserInfo[i] do
    RichEdit4.Lines.Add (Format ('User: %s, Full Name: %s, Id: %d',
      [UserName, FirstName + ' ' + LastName, UserId]));
```

Real-World Blocks

Up to now, we've discussed specific techniques related to InterBase programming, but we haven't delved into the development of an actual application, with the problems this presents in practice. In the following subsections, I'll discuss a few practical techniques, with no specific order.

Nando Dessena (who knows InterBase much better than I do) and I have used all of these techniques in a seminar discussing the porting of an internal Paradox application to InterBase. The application discussed in that circumstance was much larger and more complex, and I've trimmed it down to only a few tables to make it fit into the space I have for this chapter.

TIP The database discussed in this section is called `mastering.gdb` and is hosted on the companion CD inside the `data` subfolder of the folder for this chapter. You can examine it using InterBase Console, possibly after making a copy to a writable drive so that you can fully interact with it.

Generators and IDs

I've mentioned in the last chapter that I'm quite a fan of an extensive use of IDs to identify the records in each table of a database.

NOTE I even tend to use a single sequence of IDs for an entire system, something often indicated as an Object ID (OID). The advantage is that I can place a series of related objects in different tables, depending on their internal structure, one of the possible approaches for implementing inheritance using relational tables. In such a circumstance, however, the IDs of the two tables must be unique. As you might not know in advance which objects could be used in place of others, adopting a global OID allows more freedom later. The drawback is that, if you have lots of data, using an integer as the ID (that is, having only 4 billion objects) might not be enough. For this reason, InterBase 6 supports 64-bit generators.

How do you generate the unique values for these IDs when multiple clients are running? Keeping a table with a *latest* value is going to create troubles, as multiple concurrent transactions (from different users) will see the same values. If you don't use tables, you can use a database-independent mechanism, including the rather large Windows GUIDs or the so-called high-low technique (the assignment of a base number to each client at startup—the high number—that is combined with a consecutive number—the low number—determined by the client).

Another approach, bound to the database, is the use of internal mechanisms for sequences, indicated with different names in each SQL server. In InterBase they are called *generators*. The characteristic of these sequences is that they operate and are incremented outside of transactions, so that they provide unique numbers even to concurrent users (remember that InterBase forces you to open a transaction even to read data).

We've already seen how to create a generator. Here is the definition for the one in my demo database, followed by the definition of the view you can use to query for a new value:

```
create generator g_master;

create view v_next_id (
  next_id
  ) as
select gen_id(g_master, 1) from rdb$database
;
```

Inside the RWBlocks application, I've added to a data module an IBQuery component (as I don't need it to be an editable dataset) with the following SQL:

```
select next_id from v_next_id;
```

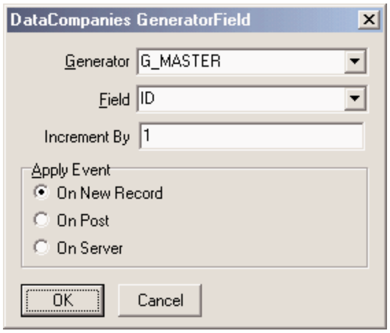
The advantage, compared to using the direct statement, is that this is easier to write and maintain, even if the underlying generator changes (or in case you switch to a different approach behind the scenes). Moreover, in the same data module I've added a function, which returns a new value for the generator:

```
function TDmMain.GetNewId: Integer;
begin
  // return the next value of the generator
  QueryId.Open;
  try
    Result := QueryId.Fields[0].AsInteger;
  finally
    QueryId.Close;
  end;
end;
```

This method can be called in the `AfterInsert` event of any dataset, to fill in the value for the ID:

```
mydataset.FieldByName ('ID').AsInteger := data.GetNewId;
```

As I've mentioned, the IBX datasets in Delphi 6 can be tied directly to a generator, simplifying the overall picture quite a lot. Thanks to the specific property editor (shown in Figure 15.8), in fact, connecting a field of the dataset to the generator becomes trivial.



DataCompanies GeneratorField

Generator: G_MASTER

Field: ID

Increment By: 1

Apply Event

● On New Record

○ On Post

○ On Server

FIGURE 15.8: The editor of the `GeneratorField` property of the IBX datasets in Delphi 6

Notice that both these approaches are much better than the one, based on a server-side trigger, discussed earlier in this chapter. In that case, in fact, the Delphi application didn't know the ID of the record sent to the database and so was unable to refresh it. Not having the record ID (which is also the only key field) on the Delphi side implies it is almost impossible to insert such a value directly inside a DBGrid. If you try, you'll see that the value you insert apparently gets lost right away, only to reappear in case of a full refresh.

Using client-side techniques instead, based on the manual code or the `GeneratorField` property, causes no trouble, as the Delphi application knows the ID, the record key, before posting it, so it can easily place it in a grid and refresh it properly.

Case-Insensitive Searches

An interesting issue with SQL servers in general, not specifically InterBase, has to do with case-insensitive searches. Suppose you don't want to show a large amount of data inside a grid (which is rather a bad idea for a client/server application). You instead choose to let the user type the initial portion of a name and then filter a query on this input, displaying only the smaller resulting record set in a grid. I've done this for a table of companies.

This search by company name is going to be executed quite frequently and will probably take place on a large table. However, if we simply search using the `starting with` or `like` operators, the search will be case sensitive, as in the following SQL statement:

```
select * from companies
where name starting with 'win';
```

To make a case-insensitive search, you can use the `upper` function on both sides of the comparison to test the uppercase values of each string, but a similar query would be very slow, as it won't be based on an index. On the other hand, saving the company names (or any other name) in uppercase letters would be rather silly, because when you have to print out those names, the result will be quite unnatural (even if very common in old information systems).

If we can trade off some disk space and memory for the extra speed, we can use a trick: add an extra field to the table, to store the uppercase value of the company name, using a server-side trigger to generate it and update it. We can then ask the database to maintain an index on the uppercase version of the name, to speed our search operation even further.

In practice, the table definition will look like this:

```
create domain d_uid as integer;
create table companies
(
  id          d_uid not null,
  name        varchar(50),
  tax_code    varchar(16),
  name_upper  varchar(50),
constraint companies_pk primary key (id)
);
```

To copy the uppercase name of each company into the related field, we cannot rely on client-side code, as an inconsistency would cause problems. In a case like this, it is better to use a trigger on the server, so that each time the company name changes, its uppercase version is updated accordingly. Another trigger will be used for the insertion of a new company:

```
create trigger companies_bi for companies
active before insert position 0
as
begin
  new.name_upper = upper(new.name);
end;

create trigger companies_bu for companies
active before update position 0
as
begin
  if (new.name <> old.name) then
    new.name_upper = upper(new.name);
end;
```

Finally, I've added an index to the table with this DDL statement:

```
create index i_companies_name_upper on companies(name_upper);
```

With this structure behind the scenes, we can now select all the companies starting with the text of an edit box (edSearch) by writing the following code in a Delphi application:

```
dm.DataCompanies.Close;
dm.DataCompanies.SelectSQL.Text :=
  'select c.id, c.name, c.tax_code,' +
    '  from companies c ' +
    '  where name_upper starting with ''' +
    UpperCase (edSearch.Text) + '''';
  dm.DataCompanies.Open;
```

TIP Using a prepared parametric query, we might be able to make this code even faster.

As an alternative, we could have created a server-side calculated field in the table definition, but this would have prevented us from having an index on the field, which speeds up our queries considerably:

```
name_upper  varchar(50) computed by (upper(name))
```

Handling Locations and People

You might notice that the table describing companies is quite bare. In fact, it has no company address, nor any contact information. The reason is simple: I want to be able to handle companies that have multiple offices (or locations) and list contact information about multiple employees of those companies.

Every location is bound to a company. Notice, though, that I've decided not to use a location identifier related to the company (such as a progressive location number for each company) but a global ID for all of the locations. This way I can refer to a location ID (let's say, for shipping goods) without having to refer also to the company ID. This is the definition of the table storing company locations:

```
create table locations
(
  id          d_uid not null,
  id_company  d_uid not null,
  address     varchar(40),
  town        varchar(30),
  zip         varchar(10),
  state       varchar(4),
  phone       varchar(15),
  fax         varchar(15),
constraint locations_pk primary key (id),
constraint locations_uc unique (id_company, id)
);

alter table locations add constraint locations_fk_companies
  foreign key (id_company) references companies (id)
  on update no action on delete no action;
```

The final definition of a foreign key relates the `id_company` field of the locations table with the `ID` field of the companies table. The other table lists names and contact information for people at specific company locations. To follow the database normalization rules, I should have added to this table only a reference to the location, as each location relates to a company. However, to make it simpler to change the location of a person within a company and to make my queries much more efficient (avoiding an extra step), I've added to the people table both a reference to the location and to the company.

The table also has another unusual feature: One of the people working for a company can be set as the key contact. This is obtained with a Boolean field (defined with a domain, as the Boolean type is not supported by InterBase) and by adding triggers to the table so that only one employee of each company can have this flag active:

```
create domain d_boolean as char(1)
  default 'F'
  check (value in ('T', 'F')) not null

create table people
(
  id           d_uid not null,
  id_company   d_uid not null,
  id_location  d_uid not null,
  name         varchar(50) not null,
  phone        varchar(15),
  fax          varchar(15),
  email        varchar(50),
  key_contact  d_boolean,
constraint people_pk primary key (id),
constraint people_uc unique (id_company, name)
);

alter table people add constraint people_fk_companies
  foreign key (id_company) references companies (id)
  on update no action on delete cascade;
alter table people add constraint people_fk_locations
  foreign key (id_company, id_location)
  references locations (id_company, id);

create trigger people_ai for people
active after insert position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;
```

```
create trigger people_au for people
active after update position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T' and old.key_contact = 'F') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;
```

Building a User Interface

The three tables we have discussed so far have a clear master/detail relation. For this reason, the RWBlocks example uses three IBDataSet components for accessing the data, hooking up the two secondary tables to the main one. The code for the master/detail support is that of a standard database example based on queries, so I won't discuss it further (but I suggest you study the source code of the example).

Each of the datasets has a full set of SQL statements, to make the data editable. Whenever you enter a new detail element, the program hooks it to its master tables, as in the two following methods:

```
procedure TDmCompanies.DataLocationsAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataLocationsID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
end;

procedure TDmCompanies.DataPeopleAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataPeopleID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
  // the suggested location is the active one, if available
  if not DataLocations.IsEmpty then
    DataPeopleID_LOCATION.AsInteger := DataLocationsID.AsInteger;
  // the first person added becomes the key contact
  // (checks whether the filtered dataset of people is empty)
  DataPeopleKEY_CONTACT.AsBoolean := DataPeople.IsEmpty;
end;
```

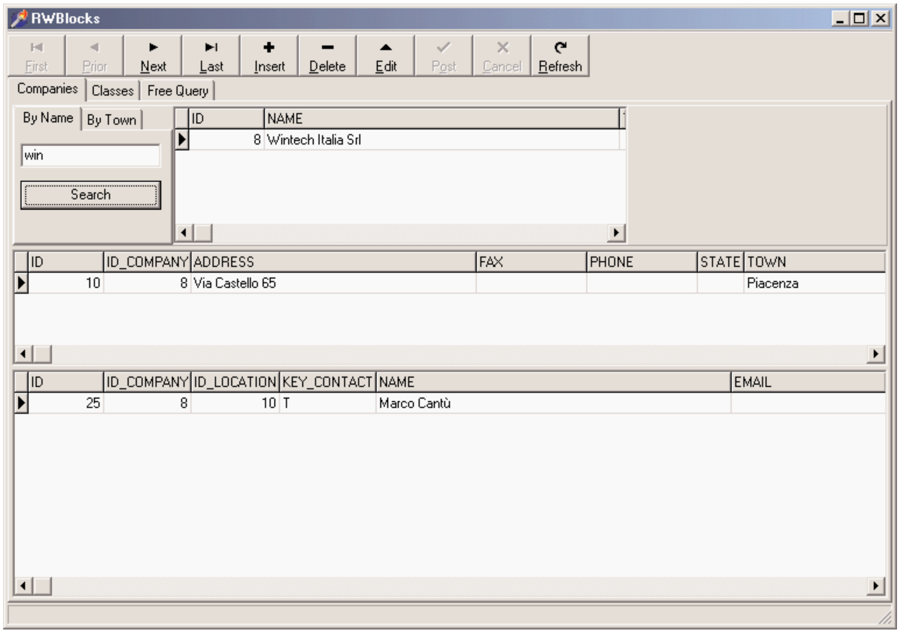
As this code suggests, a data module hosts the dataset components. Actually, the program has a data module for every form (hooked up dynamically, as you can create multiple instances of each form). Each of these data modules has a separate transaction, so that the various operations done in different pages are totally independent. The database connection, instead, is centralized. A main data module hosts the corresponding component, which is referenced by all the datasets. Each of the data modules is created dynamically by the form referring to it, and its value is stored in the dm private field of the form:

```
procedure TFormCompanies.FormCreate(Sender: TObject);
begin
  dm := TDmCompanies.Create (Self);
  dsCompanies.Dataset := dm.DataCompanies;
  dsLocations.Dataset := dm.DataLocations;
  dsPeople.Dataset := dm.DataPeople;
end;
```

This way we can easily create multiple instances of a form, with an instance of the data module connected to each of them. The form connected to the data module has three DBGrid controls, each tied to a data module and one of the corresponding datasets. You can see this form at run time, with some actual data, in Figure 15.9.

FIGURE 15.9:

A form showing companies, office locations, and people (part of the RWBlocks example)



win

ID	NAME
8	Wintech Italia Srl

ID	ID_COMPANY	ADDRESS	FAX	PHONE	STATE	TOWN
10	8	Via Castello 65				Piacenza

ID	ID_COMPANY	ID_LOCATION	KEY_CONTACT	NAME	EMAIL
25	8	10	T	Marco Cantù	

The form is actually hosted by a main form, which in turn is based on a page control, with the other forms embedded. Only the form connected with the first page is created when the program starts. The ShowForm method I've written takes care of parenting the form to the tab sheet of the page control, after removing the form border:

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  ShortDateFormat := 'dd/mm/yyyy';
  ShowForm (TFormCompanies.Create (self), TabCompanies);
end;

procedure TFormMain.ShowForm (Form: TForm; Tab: TTabSheet);
begin
  Form.BorderStyle := bsNone;
  Form.Align := alClient;
  Form.Parent := Tab;
  Form.Show;
end;
```

The other two pages, instead, are populated at runtime:

```
procedure TFormMain.PageControl1Change(Sender: TObject);
begin
  if PageControl1.ActivePage.ControlCount = 0 then
    if PageControl1.ActivePage = TabFreeQ then
      ShowForm (TFormFreeQuery.Create (self), TabFreeQ)
    else if PageControl1.ActivePage = TabClasses then
      ShowForm (TFormClasses.Create (self), TabClasses);
end;
```

The companies form hosts the search by company name we've already discussed in the last section, plus a search by location. You enter the name of a town and get back a list of companies having an office in that town:

```
procedure TFormCompanies.btnTownClick(Sender: TObject);
begin
  with dm.DataCompanies do
  begin
    Close;
    SelectSQL.Text :=
      'select c.id, c.name, c.tax_code' +
      '  from companies c ' +
      '  where exists (select loc.id from locations loc ' +
      '  where loc.id_company = c.id and upper(loc.town) = ''' +
      UpperCase(edTown.Text) + ''' )';
    Open;
    dm.DataLocations.Open;
    dm.DataPeople.Open;
  end;
end;
```

If you look at the source code of the form, you'll find a lot more code. Some of it is related to closing permission (as a user cannot close the form while there are pending edits not posted to the database), while a good amount relates to the use of the form as a lookup dialog, as described later.

Booking Classes

Another portion of the program and of the database involves booking training classes and courses. (Needless to say, although I built this program as a showcase, it also helps me run my own business.) In the database is a classes table listing all the training courses, each with a title and the planned date. Another table hosts registration by company, including the classes registered for, the ID of the company, and some notes. Finally, a third table has a list of people who've signed up, each hooked to a registration for his or her company, with the amount paid.

The rationale behind this company-based registration is that invoices are sent out to companies, which book the classes for their programmers and can receive specific discounts. This is a case in which the database is a little more normalized, as the people registration doesn't refer directly to a class, but only to the company registration for that class. Here is the definition of the tables involved (I've omitted foreign key constraints and other elements):

```
create table classes
(
  id           d_uid not null,
  description  varchar(50),
  starts_on    timestamp not null,
constraint classes_pk primary key (id)
);
create table classes_reg
(
  id          d_uid not null,
  id_company  d_uid not null,
  id_class    d_uid not null,
  notes       varchar(255),
constraint classes_reg_pk primary key (id),
constraint classes_reg_uc unique (id_company, id_class)
);
create domain d_amount as numeric(15, 2);
create table people_reg
(
  id              d_uid not null,
  id_classes_reg  d_uid not null,
  id_person       d_uid not null,
  amount          d_amount,
constraint people_reg_pk primary key (id)
);
```

The data module for this group of tables uses a master/detail/detail relationship, and has code to set the connection with the active master record when a new detail record is created. Each dataset has a generator field for its ID, and each has the proper `update` and `insert` SQL statements. These statements have been generated by the corresponding component editor using only the ID field to identify existing records and updating only the fields of the original table. In fact, each of the two secondary datasets retrieves data from a lookup table, either the list of companies or the list of people. Finally, I had to edit manually the `RefreshSQL` statements to repeat the proper inner join. Here is an example:

```
object IBClassReg: TIBDataSet
  Database = DmMain.IBDatabase1
  Transaction = IBTransaction1
  AfterInsert = IBClassRegAfterInsert
  DeleteSQL.Strings = (
    'delete from classes_reg'
    'where id = :old_id')
  InsertSQL.Strings = (
    'insert into classes_reg (id, id_class, id_company, notes)'
    'values (:id, :id_class, :id_company, :notes)')
  RefreshSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id = :id')
  SelectSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id_class = :id')
  ModifySQL.Strings = (
    'update classes_reg'
    'set'
    '  id = :id,'
    '  id_class = :id_class,'
    '  id_company = :id_company,'
    '  notes = :notes'
    'where id = :old_id')
  GeneratorField.Field = 'id'
  GeneratorField.Generator = 'g_master'
  DataSource = dsClasses
end
```

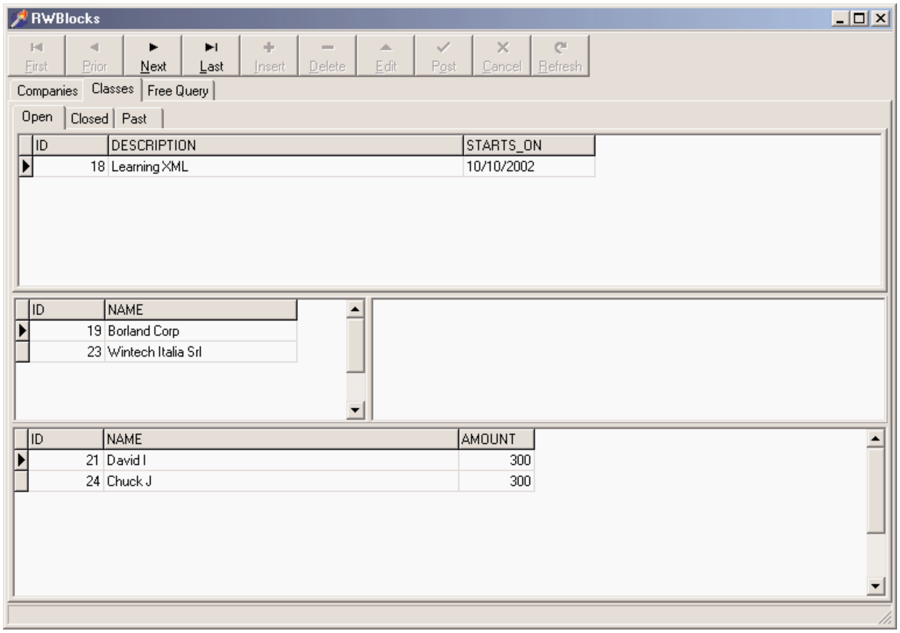
To complete the discussion of `IBClassReg`, here is its only event handler:

```
procedure TDmClasses.IBClassRegAfterInsert(DataSet: TDataSet);
begin
  IBClassReg.FieldByName ('id_class').AsString :=
    IBClasses.FieldByName ('id').AsString;
end;
```

The IBPeopleReg dataset has similar settings, but the IBClasses dataset is simpler, at design time. At run time, the SQL code of this dataset is dynamically modified, using three alternatives to display scheduled classes (whenever the date is after today's date), classes already started or finished in the current year, and classes of past years. A user selects one of the three groups of records for the table with a tab control, which hosts the DBGrid for the main table (see Figure 15.10).

FIGURE 15.10:

The form for class registrations of the RWBlocks example



ID	DESCRIPTION	STARTS_ON
18	Learning XML	10/10/2002

ID	NAME
19	Borland Corp
23	Wintech Italia Srl

ID	NAME	AMOUNT
21	David I	300
24	Chuck J	300

The three alternative SQL statements are created when the program starts, or actually when the class registrations form is created and displayed. The program stores the final portion of the three alternative instructions (the where clause) in a string list, and selects one of the strings when the tab changes:

```
procedure TFormClasses.FormCreate(Sender: TObject);
begin
  dm := TDmClasses.Create (Self);
  // connect the datasets to the data sources
  dsClasses.Dataset := dm.IBClasses;
  dsClassReg.DataSet := dm.IBClassReg;
  dsPeopleReg.DataSet := dm.IBPeopleReg;
  // open the datasets
  dm.IBClasses.Active := True;
  dm.IBClassReg.Active := True;
  dm.IBPeopleReg.Active := True;
  // prepare the SQL for the three tabs
  SqlCommands := TStringList.Create;
  SqlCommands.Add (' where Starts_On > ''now''');
  SqlCommands.Add (' where Starts_On <= ''now'' and ' +
    ' extract (year from Starts_On ) >= extract(year from current_timestamp)');
  SqlCommands.Add (' where extract (year from Starts_On) < ' +
    ' extract(year from current_timestamp)');
end;
procedure TFormClasses.TabChange(Sender: TObject);
begin
  dm.IBClasses.Active := False;
  dm.IBClasses.SelectSQL [1] := SqlCommands [Tab.TabIndex];
  dm.IBClasses.Active := True;
end;
```

Building a Lookup Dialog

The two detail datasets of this class registration form display some lookup fields. Instead of showing the ID of the company that booked the class, for example, it shows the company name. This is obtained with an inner join in the SQL statement and by configuring the DBGrid columns not to display the company ID. In a local application, or one with a limited amount of data, we could have used a lookup field. However, copying the entire lookup dataset locally or opening it for browsing should be limited to tables with about a hundred records at most, embedding some search capabilities.

If you have a large table, like a table of companies, an alternative solution can be to use a secondary dialog box to do the lookup selection. For example, we can choose a company using the form we've already built and taking advantage of its search capabilities. To display this form as a dialog box, the program creates a new instance of it, shows some hidden buttons already there at design time, and lets the user select a company to refer to from the other table.

To simplify the use of this lookup, which can happen multiple times in a large program, I've added to the companies form a class function, having as output parameters the name and ID of the selected company. An initial ID can be passed to the function to determine its initial selection. Here is the complete code of this class function, which creates an object of its class, selects the initial record if requested, shows the dialog box, and finally extracts the return values:

```
class function TFormCompanies.SelectCompany (
  var CompanyName: string; var CompanyId: Integer): Boolean;
var
  FormComp: TFormCompanies;
begin
```

```
    Result := False;
    FormComp := TFormCompanies.Create (Application);
    FormComp.Caption := 'Select Company';
    try
      // activate dialog buttons
      FormComp.btnCancel.Visible := True;
      FormComp.btnOK.Visible := True;
      // select company
      if CompanyId > 0 then
        FormComp.dm.DataCompanies.SelectSQL.Text :=
          'select c.id, c.name, c.tax_code' +
          '   from companies c ' +
          '   where c.id = ' + IntToStr (CompanyId)
      else
        FormComp.dm.DataCompanies.SelectSQL.Text :=
          'select c.id, c.name, c.tax_code' +
          '   from companies c ' +
          '   where name_upper starting with ''a''';
      FormComp.dm.DataCompanies.Open;
      FormComp.dm.DataLocations.Open;
      FormComp.dm.DataPeople.Open;

      if FormComp.ShowModal = mrOK then
      begin
        Result := True;
        CompanyId := FormComp.dm.DataCompanies.FieldByName ('id').AsInteger;
        CompanyName := FormComp.dm.DataCompanies.FieldByName ('name').AsString;
      end;
    finally
      FormComp.Free;
    end;
  end;
```

Another slightly more complex class function (available within the example's source code, but not listed here) allows the selection of a person of a given company to register people for classes. In this case, the form is displayed after disallowing searching another company or changing it.

In both cases, the lookup is triggered by adding an ellipsis button to the column of the DBGrid—for example, the grid column listing the names of companies registered for classes. When this button is pressed, the program calls the class function to display the dialog box and uses its result for updating the hidden ID field and the visible name field:

```
  procedure TFormClasses.DBGridClassRegEditButtonClick(Sender: TObject);
  var
    CompanyName: string;
    CompanyId: Integer;
  begin
    CompanyId := dm.IBClassReg.FieldByName ('id_Company').AsInteger;
    if TFormCompanies.SelectCompany (CompanyName, CompanyId) then
    begin
      dm.IBClassReg.Edit;
      dm.IBClassReg.FieldByName ('Name').AsString := CompanyName;
      dm.IBClassReg.FieldByName ('id_Company').AsInteger := CompanyId;
    end;
  end;
```

Adding a Free Query Form

The final feature of the program is a form where a user can directly type in a SQL statement and run it. As a helper, the form lists in a combo box the available tables of the database, obtained when the form is created by calling:

```
DmMain.IBDatabase1.GetTableNames (ComboTables.Items);
```

Selecting an item of the combo box generates a simple SQL query:

```
MemoSql.Lines.Text := 'select * from ' + ComboTables.Text;
```

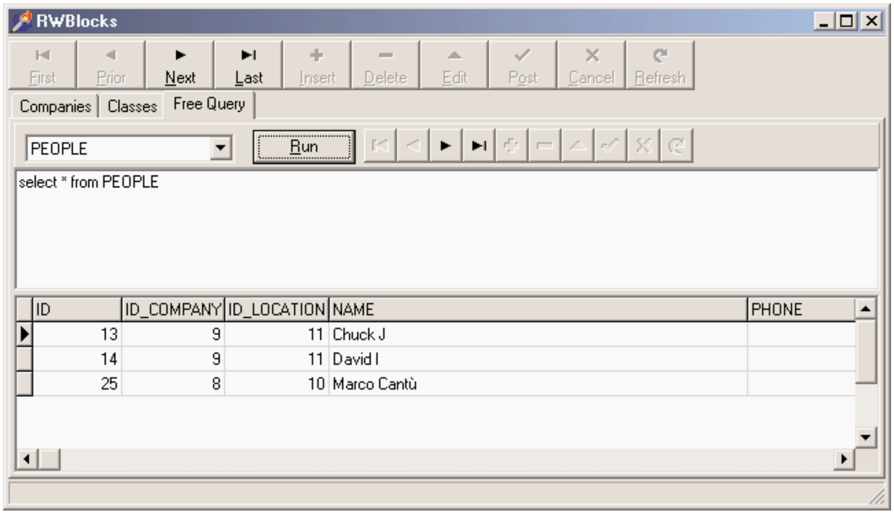
The user, if an expert, can then edit the SQL, possibly introducing restrictive clauses, and then run the query:

```
procedure TFormFreeQuery.ButtonRunClick(Sender: TObject);
begin
  QueryFree.Close;
  QueryFree.SQL := MemoSql.Lines;
  QueryFree.Open;
end;
```

You can see this third form of the RWBlocks program in Figure 15.11. Of course, I'm not suggesting that you add SQL editing to programs intended for all of your users. This feature is intended for power users, maybe programmers. I basically wrote it for myself!

FIGURE 15.11:

The free query form of the RWBlocks example is intended for power users.



```
select * from PEOPLE
```

ID	ID_COMPANY	ID_LOCATION	NAME	PHONE
13	9	11	Chuck J	
14	9	11	David I	
25	8	10	Marco Cantù	

What's Next?

After looking at other, more general, database access technologies Delphi provides (such as the BDE and dbExpress), in this chapter I've introduced the InterBase database and the IBX family of components. The last part of the chapter presented a complete real-world application, discussing a series of general-purpose techniques you can probably apply even to completely different InterBase applications.

Now we are ready to focus on another data access alternative, Microsoft's own ADO technology, which Delphi fully supports with a specific set of dataset components since version 5. In subsequent chapters, we'll continue to explore database development, with multitier architectures and the development of database-oriented Delphi components.

CHAPTER 16

ActiveX Data Objects

- Microsoft Data Access Components (MDAC)
- dbGo
- Data link files
- Getting schema information
- Using the Jet engine
- Transaction processing
- Disconnected and persistent recordsets
- The briefcase model and deploying MDAC

I wish to acknowledge and thank Guy Smith-Ferrier for writing this chapter. Guy is a programmer, author, and speaker. He is the author of several commercial software products and countless internal systems for independent and blue-chip companies alike. He has written many articles for *The Delphi Magazine* and for others on topics beyond Delphi, and he has spoken at numerous conferences in North America and Europe. Guy lives in England with his wife, his son, and his cat.

Since the mid-1980s, database programmers have been on a quest for the "holy grail" of *database independence*. The idea is to use a single API that applications can use to interact with many different sources of data. The use of such an API would release developers from a dependence upon a single database engine and allow them to adapt to the world's changing demands. Vendors have produced many solutions to this goal, the two most notable early solutions being Microsoft's Open Database Connectivity (ODBC) and Borland's Independent Database Application Programming Interface (IDAPI), more commonly known as the Borland Database Engine (BDE).

Microsoft started to replace ODBC with OLE DB in the mid-1990s with the success of COM. However, OLE DB is what Microsoft would class a *system-level* interface and is intended to be used by system-level programmers. It is very large and complex and unforgiving. It makes greater demands on the programmer and requires a higher level of knowledge in return for lower productivity. ActiveX Data Objects (ADO) is a layer on top of OLE DB and is referred to as an *application-level* interface. It is considerably simpler than OLE DB and more forgiving. In short, it is designed for use by application programmers.

ADO has great similarities with the BDE. They are, after all, designed to solve very similar problems. Both support navigation of datasets, manipulation of datasets, transaction processing, and cached updates (called *batch updates* in ADO), so the concepts and issues involved in using ADO are similar to those of the BDE. However, there are also differences. ADO is a more recent technology. This gives it an advantage over the BDE because it is better suited to today's needs and doesn't need to carry so much deadwood around with it. Perhaps more importantly, ADO has a wider interpretation of "data." The BDE is used for accessing "rectangular" data—that is, data in rows and columns. This is ideal for accessing data from databases. ADO can be used for accessing this data but can also be used for accessing non-rectangular data, including directory structures, documents, Web sites, and e-mail.

In this chapter we will look at ADO and *dbGo*. (This set of Delphi components was called ADOExpress but has been renamed dbGo in Delphi 6, because Microsoft has objected to the use of the term *ADO* in product names.) It is possible to use ADO in Delphi without using dbGo. By importing the ADO type library, you can gain direct access to the ADO interfaces; this is, indeed, how Delphi programmers used ADO before the release of Delphi 5. However, this path bypasses Delphi's database infrastructure and ensures that you are unable to make use of other Delphi technologies such as DataSnap. Alternatively, you can turn to Delphi's active third-party market for other ADO component suites such as Adonis, AdoSolutio, Diamond ADO, and Kamiak.

This chapter uses dbGo for all of its examples, not only because it is readily available and supported but also because it is a very viable solution. Regardless of your final choice, you will find the information here useful.

Microsoft Data Access Components (MDAC)

ADO is part of a bigger picture called Microsoft Data Access Components (MDAC). MDAC is an umbrella for Microsoft's database technologies and includes ADO, OLE DB, ODBC, and RDS (Remote Data Services). Often you will hear people use the terms MDAC and ADO interchangeably (but incorrectly) because their version numbers and releases are now aligned. As ADO is only distributed as part of MDAC, we talk in terms of MDAC releases. The major releases of MDAC have been versions 1.5, 2.0, 2.1, 2.5, and 2.6. Microsoft releases MDAC independently and makes it available for free download and virtually free distribution (there are distribution requirements, but almost all Delphi developers will not have trouble meeting these requirements). MDAC is also distributed with most Microsoft products that have some kind of database content. This includes Windows 98, Windows 2000, Windows Millennium Edition, Microsoft Office, Internet Explorer, and SQL Server. In addition, Delphi 6 Enterprise ships with MDAC 2.5, and Delphi 5 Enterprise ships with MDAC 2.1.

There are two consequences of this level of availability. First, it is highly likely that your users will already have MDAC installed on their machines. Second, whatever version your users have, or you upgrade them to, it is also virtually certain that someone, either you, your users, or other application software, will upgrade their existing MDAC to whatever the current release of MDAC is. There is no way you will be able to prevent this, as MDAC is installed with such commonly used software as Internet Explorer. Add to this the fact that Microsoft supports only the current release of MDAC and the release before it, and you are forced to arrive at a conclusion:

> Your applications must be designed to work with the current release of MDAC or the release before it.

If you chart the releases of MDAC, you can expect to see a new version of MDAC every 10 months on average (Delphi itself has a new release every 14 months on average).

As an ADO developer, you should regularly check the MDAC pages on Microsoft's site at `www.microsoft.com/data`. From there you can download the latest version of MDAC for free. At the time of writing, this is MDAC 2.6, which you can download from `www.microsoft.com/data/download_260rtm.htm` (5.2 MB), but you should check for a more recent version first.

While you are on this Web site, you should take the opportunity to download the MDAC SDK (13 MB) if you do not already have it or the Platform SDK (the MDAC SDK is part of the Platform SDK). The MDAC SDK is your bible. Download it and consult it regularly and use it to answer your ADO questions. You should treat this as your first port of call when you need MDAC information. Beyond this, you should read the README files that come with MDAC. Look in `\Program Files\Common Files\System\ADO` for all of the files ending `README.TXT`.

Finally, in getting ready for using ADO in your Delphi applications, you should check for dbGo/ADOExpress updates on Borland's excellent community site (`http://community.borland.com`). Informal patches are released here that are often a must-have. For example, there is a show-stopping problem when using MDAC 2.6 and Delphi 5's ADOExpress, which requires a patch.

OLE DB Providers

OLE DB providers enable access to a source of data. They are ADO's equivalent to the BDE's drivers. However, although the BDE's Driver SDK has been available for many years, there are no third-party BDE drivers. This is not the case for OLE DB providers. MDAC includes many providers that I'll discuss, but many more are available from Microsoft and, more prolifically, the third-party market. It is no longer possible to reliably list all available OLE DB providers, because the list is so large and ever-changing, but here are some of the main vendors of these drivers:

Company	Web Site	OLE DB Provider
B2 Systems	`www.b2systems.com`	SQL Server, Oracle, Sybase, Informix, RDB, DB2, flat files
ISG	`www.isgsoft.com`	ISG Navigator (ISAM, DB2, IMS, Informix, Jasmine, Open Ingres, Oracle, SQL Server, Sybase, Adabas, RDB, RMS, VSAM)
Merant	`www.merant.com`	DB2, Informix, Lotus Notes, SQL Server, Ingres, Oracle, Sybase

You should add to this list almost all database vendors, as the majority now supply their own OLE DB providers. For example, Oracle supplies the ORAOLEDB provider. Notable omissions include InterBase; at the time of writing, Borland doesn't plan to write an OLE DB provider for InterBase. Your only solutions are to access InterBase either using the ODBC Driver, or through Jason Wharton's InterBase provider (`www.ibobjects.com`, although this is still in development) or Dmitry Kovalenko's IBProvider (`www.lcpi.lipetsk.ru/prog/eng/index.html`).

TIP Jason Wharton's OLE DB Provider is additionally interesting because it is written using Binh Ly's OLE DB Provider Development Toolkit (`www.techvanguards.com/products/optk/install.htm`). If you want to write your own OLE DB provider, this is an easier way than most.

MDAC OLE DB Providers

When you install MDAC, you automatically install the OLE DB providers shown in Table 16.1:

TABLE 16.1: OLE DB Providers Included with MDAC

Driver	Provider	Description
MSDASQL	ODBC Drivers	ODBC drivers (default)
Microsoft.Jet.OLEDB.3.5	Jet 3.5	MS Access 97 databases only
Microsoft.Jet.OLEDB.4.0	Jet 4.0	MS Access databases et al.
SQLOLEDB	SQL Server	MS SQL Server databases
MSDAORA	Oracle	Oracle databases
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	Example of an OLE DB provider for CSV files
MSDAOSP	Simple provider	For creating your own providers for simple text data

If you do not specify which OLE DB provider you are using, OLE DB defaults to the ODBC OLE DB Provider, which is used for backward compatibility with ODBC. As you learn more about ADO, you will discover the limitations of this provider. It's probable that you will eventually tire of these limitations, and I recommend that you look from the beginning for an OLE DB provider that is specific to your database rather than struggle with the ODBC OLE DB Provider.

The Jet OLE DB Providers support MS Access and other "desktop" databases. We will return to these providers later.

The SQL Server Provider supports SQL Server 7, SQL Server 2000, and Microsoft Database Engine (MSDE). MSDE is worth taking a moment's thought over. MSDE is SQL Server with most of the tools removed and some code added to deliberately degrade performance when there are more than 5 active connections. MSDE is important for two reasons. First, it is free. You can download it from Microsoft's Web site and, with very few restrictions, distribute it with your application. Second, MSDE is SQL Server. Of course you don't get the

SQL Server tools, and it does deliberately degrade performance, but it is SQL Server. This means that it is perfect for use with low numbers of users. When the number of users increases and performance starts to suffer, your upgrade path to SQL Server is just a question of paying for SQL Server. Compatibility is virtually assured. You use the same OLE DB provider, and you use it in exactly the same way with exactly the same names and parameters. This is because MSDE is SQL Server. Because of these reasons and because Microsoft is moving their emphasis away from Access, MSDE is worth considering for future developments.

The OLE DB Provider For OLAP can be used directly but is more often used by ADO Multi-Dimensional (ADOMD). ADOMD is an additional ADO technology designed to provide Online Analytical Processing (OLAP). If you have used Delphi's Decision Cube, or Excel's Pivot Tables, or Access's Cross Tabs, then you have used some form of OLAP.

In addition to these MDAC OLE DB providers, Microsoft supplies other OLE DB providers with other products or with downloadable SDKs. The Active Directory Services OLE DB Provider is included with the ADSI SDK; the AS/400 And VSAM OLE DB Provider is included with SNA Server; and the Exchange OLE DB Provider is included with Microsoft Exchange 2000.

The OLE DB Provider For Indexing Service provides access to (and is part of) Microsoft Indexing Service, a Windows NT and 2000 mechanism that speeds up file searches by building catalogs of file information. Indexing Service is integrated into IIS and, consequently, is often used for indexing Web sites. Microsoft Indexing Service is also available for Windows NT 4 as part of the NT 4 Option Pack.

The OLE DB Provider For Internet Publishing is included with Internet Explorer 5, Windows 2000, and Office 2000 and allows developers to manipulate directories and files using HTTP. This is useful for maintaining Web sites that support either FrontPage Web Extender (WEC) or Web Distributed Authoring and Versioning (WebDAV).

Still more OLE DB providers come in the form of *service providers*. As their name implies, OLE DB service providers provide a service to other OLE DB providers. Often these service providers will go unnoticed because they are invoked automatically as needed without programmer intervention. The Cursor Service, for example, is invoked when you create a client-side cursor, and the Persisted Recordset provider is invoked to save data locally.

dbGo

The set of components that make up dbGo (Table 16.2) should be easily recognizable by programmers familiar with the BDE, dbExpress, or IBExpress.

TABLE 16.2: dbGo Components

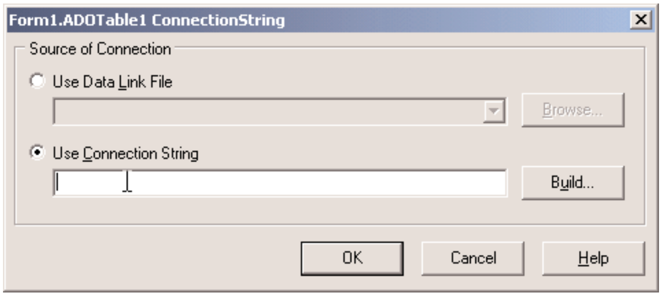
dbGo Component	Description	BDE Equivalent Component
TADOConnection	Connection to a database	TDatabase
TADOCommand	Executes an action SQL command	No equivalent
TADODataSet	All-purpose TDataSet	No equivalent
TADOTable	Encapsulation of a table	TTable
TADOQuery	Encapsulation of SQL SELECT	TQuery
TADOStoredProc	Encapsulation of a stored procedure	TStoredProc
TRDSConnection	Remote Data Services connection	No equivalent

The four dataset components (TADODataSet, TADOTable, TADOQuery, TADOStoredProc) are implemented almost entirely by their immediate ancestor TCustomADODataSet. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component. As such, the components have a lot in common. In general, however, TADOTable, TADOQuery, and TADOStoredProc are viewed as "compatibility" components and are used to aid the transition of knowledge and code from their BDE counterparts. Be warned, though: These compatibility components are similar to their counterparts but not exactly the same. You will find differences in any application except the most trivial. TADODataSet is the component of choice partly because of its versatility but also because it is closer in appearance to the ADO Recordset interface upon which it is based. Throughout this chapter, we will use all of the TDataSet components to give you the experience of using each.

Enough theory. Let's see some action. Drop a TADOTable onto a form. Look in the Object Inspector and you will not see any `DatabaseName` or `AliasName` properties. ADO doesn't use aliases, so there are no alias-related properties. Instead ADO runs on *connection strings*. Connection strings are the lifeblood of ADO, and you should take time out to master this subject. You can type in a connection string by hand if you know what you are doing, but only programmers who think that VI is a great editor of our time enjoy doing this. For the rest of us, there is the connection string editor. In the Object Inspector, click the ellipses in the `ConnectionString` property. This invokes Delphi's connection string editor (Figure 16.1).

FIGURE 16.1:

Delphi's connection string editor



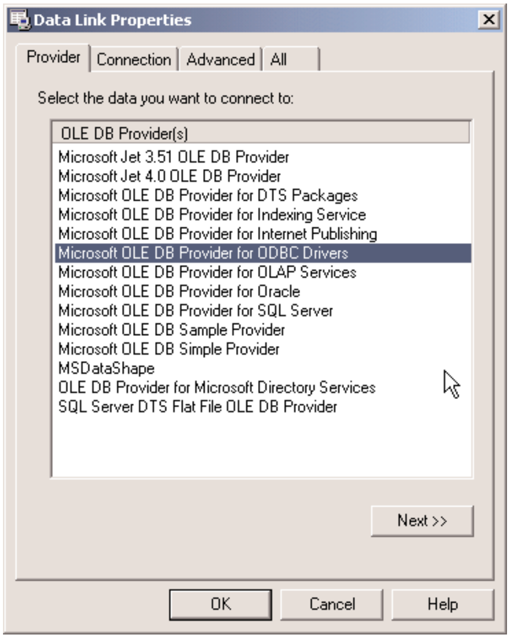
○ Use Data Link File

● Use Connection String

This editor adds little value to the process of entering a connection string, so you can click Build to go straight to Microsoft's connection string editor (Figure 16.2).

FIGURE 16.2:

Microsoft's connection string editor



Select the data you want to connect to:

OLE DB Provider(s)

Microsoft Jet 3.51 OLE DB Provider
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider for DTS Packages
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Sample Provider
Microsoft OLE DB Simple Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services
SQL Server DTS Flat File OLE DB Provider

This is a tool you will come to know and love—or maybe just to know. The first tab shows the OLE DB providers and service providers installed on your computer. The list will vary according to the version of MDAC and other software installed on your computer. You can see that the OLE DB Provider for ODBC Drivers is selected by default. In our first example, we will open the infamous MS Access Northwind database using the Jet 4.0 OLE DB Provider. Northwind is the Microsoft equivalent of DBDEMOS; it is the test data used in many examples on ADO because it is so widely available. The exact location of Northwind and its name are not fixed, but you should search your hard disk for `Northwind.mdb` or `NWind.mdb`. Microsoft SQL Server comes with a very similar version of the same database, also called Northwind.

Double-click the Jet 4.0 OLE DB Provider and you will be presented with the Connection tab. This page varies according to the provider you select, but for Jet it simply asks you for the name of the database and your login details. If you have Microsoft Office installed on your computer, then the database name will probably be

```
c:\program files\microsoft office\office\samples\northwind.mdb
```

Click the Test Connection button to test the validity of your selections.

The Advanced tab handles access control to the database, and this is where you would specify exclusive or read-only access to the database.

The All tab lists all the parameters in the connection string. The list is specific to the OLE DB provider you selected on the first page. You should make a mental note of this page, because it contains many parameters that are the answers to many problems. Click OK to close the Microsoft connection string editor, click OK again to close the Borland connection string editor, and the value will be returned to the `ConnectionString` property, which will now be set to

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\program files\microsoft office\
  office\samples\northwind.mdb;Persist Security Info=False
```

So connection strings are just a string with many parameters delimited by semicolons. If you want to add, edit, or delete any of these parameter values programmatically, you must write your own routines to find the parameter in the list and amend it appropriately.

Now that we have set the connection string, we can select a table. Drop down the list of tables using the `TableName` property in the Object Inspector. Select the Customers table and set `Active` to True. Add a TDataSource and a TDBGrid and connect them all together, and you are now using ADO.

Incidentally, if you are going to use dbGo on a permanent basis, you might benefit from a simple tip. If you followed along with the last example, you will have noticed that you had to flip back and forth between the Data Access page and the ADO page to drop a TDataSource component onto the form. If you use dbGo exclusively, then the TDataSource component is the only component you will ever need on the Data Access page. Delphi's IDE prevents you from adding the same component to multiple pages, but you can move components. To move the TDataSource component from the Data Access page to the ADO page, right-click the palette, select Properties, and drag TDataSource onto the ADO page. If, however, you use both ADO and another database technology such as the BDE, then you can simulate installing

TDataSource on multiple pages by creating a Component Template for a TDataSource and installing it on the ADO page. This is a more elegant solution than creating a TADODataSource descendant, because the component that is dropped onto the form is still a genuine TDataSource and not some other component.

TADOConnection

When a TADOTable component is used in this way, it creates its own connection component behind the scenes in the same way that the BDE components create their temporary TDatabase component. You do not have to accept the default connection it creates, and you should not accept it. Instead, you should create your own connection in the form of a TADOConnection component.

The TADOConnection component is used for many of the same purposes as the BDE's TDatabase component. It allows you to customize the login procedure, control transactions, execute action commands directly, and reduce the number of connections in an application. Using a TADOConnection is easy. Place one on a form and set its `ConnectionString` property in the same way as for the TADOTable. Alternatively, you can double-click a TADOConnection to invoke the connection string editor directly. With the `ConnectionString` set to `Northwind.mdb`, you can disable the login dialog box by setting `LoginPrompt` to False. To make use of the new connection, set `ADOTable1`'s `Connection` property to `ADOConnection1`. You will see `ADOTable1`'s `ConnectionString` property reset because `Connection` and `ConnectionString` are mutually exclusive. One of the benefits of using a TADOConnection is that the connection string is centralized instead of scattered throughout many components. Another, more important, benefit is that all of the components that share the TADOConnection share a single connection. Without your own TADOConnection, each ADO dataset uses its own connection.

Data Link Files

So a TADOConnection allows us to centralize the definition of a connection string within a form or data module. However, even though this is a worthwhile step forward from scattering the same connection string throughout all ADO datasets, it still suffers from a fundamental flaw: If you use a database engine that defines the database in terms of a filename, then the path to the database file(s) is hard-coded in the EXE. This makes for a very fragile application. The BDE uses aliases to overcome this problem; ADO uses Data Link files. A Data Link file is a connection string in an INI file. The following is an example of a Data Link file:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\program files\microsoft office\
  office\samples\northwind.mdb;Persist Security Info=False
```

Although you can give a Data Link file any extension, the recommended extension is .UDL. You can create a Data Link using any text editor, or you can right-click Windows Explorer, select New, then Text Document, rename the file with a UDL extension, and then double-click the file to invoke the Microsoft connection string editor.

To use the Data Link file, set the TADOConnection's ConnectionString to:

```
File Name=TEST.UDL
```

assuming that the file is called TEST.UDL and it is in the same directory as the EXE. You can place your Data Link files anywhere on the hard disk, but if you are looking for a common, shared location, then you can use the DataLinkDir function in ADODB.PAS:

```
ShowMessage('The Data Link directory is ' + DataLinkDir);
```

If you haven't altered MDAC's defaults, DataLinkDir will return

```
C:\Program Files\Common Files\System\OLE DB\Data Links
```

Delphi 5's ADOExpress suffers from a flaw when using data link files, which is fixed in Delphi 6's dbGo. In 5, set the Connected property of the TADOConnection to True and watch the ConnectionString property. The ConnectionString becomes the actual connection string that is in use (i.e., the one from the data link file). The problem is that when the property is streamed to the form when the application is saved, it is the active connection string that is saved. The reference to the data link file is permanently forgotten. If you change the data link file, you will not see any change to your application.

If you want to use data link files and you don't want to suffer from this problem, use the following TADOConnectionX component instead of TADOConnection:

```
TADOConnectionX = class(TADOConnection)
private
  FUDLFile: string;
protected
  procedure DoConnect; override;
published
  property UDLFile: string read FUDLFile write FUDLFile;
end;
```

The component has a UDLFile property where the UDL filename is permanently stored (you must manually set this property yourself). The class has a single method, DoConnect, which ensures that the UDL file is always read each time the connection is opened:

```
procedure TADOConnectionX.DoConnect;
begin
  if FUDLFile <> '' then
    ConnectionString:= 'File Name=' + FUDLFile;
  inherited;
end;
```

Dynamic Properties

Imagine that you are responsible for designing a new database middleware architecture. You have to reconcile two opposing goals of a single API for all databases and access to database-specific features. You could take the approach of designing an interface that is the sum of all of the features of every database ever created. Each class would have every property and method imaginable, but it would only use the properties and methods it had support for. It doesn't take much discussion to realize that this isn't a good solution. ADO has to solve these apparently mutually exclusive goals, and it does so using *dynamic properties*. Almost all ADO interfaces, and their corresponding dbGo components, have a property called Properties that is a collection of database-specific properties. These properties can be accessed by their ordinal position, like this:

```
ShowMessage(ADOTable1.Properties[1].Value);
```

But they are more usually accessed by name like this:

```
ShowMessage(ADOConnection1.Properties['DBMS Name'].Value);
```

It would be tedious to list all of the different dynamic properties for all of the different classes for all of the different OLE DB providers for all of the situations in which they are used, but to give you an idea of their importance, a typical ADO Connection or Recordset has approximately 100 dynamic properties. As we will see throughout this chapter, the answers to many ADO questions lie in dynamic properties, so keep your eyes and ears open for the ones that solve your problems. An important event, `OnRecordsetCreate`, was planned to be added to TCustomADODataSet in Delphi 6, which you may need to be aware of when using dynamic properties. (This event was not yet included as this book went to press.) `OnRecordsetCreate` is called immediately after the recordset has been created but has not been opened. This is useful when setting some dynamic properties as certain properties can only be set when the recordset is closed.

Getting Schema Information

One of the BDE components for which there is no apparent ADO alternative is TSession. TSession is used for several purposes, but a common use is to retrieve schema information (information about the structure of the database and its contents). In ADO this information can be retrieved using TADOConnection's `OpenSchema` method. This method accepts four parameters. The first, and most interesting, is the kind of data that `OpenSchema` should return. It is a `TSchemaInfo` value, which is a set of 40 values including those for retrieving a list of tables, indexes, columns, views, and stored procedures. The second parameter is a filter to place on the data before it is returned. We will see an example of this parameter in a moment.

The third parameter is a GUID for a provider-specific query and is only used if the first parameter is siProviderSpecific. The fourth and final parameter is a TADODataSet into which the data is returned. This last parameter illustrates a common theme in ADO: any method that needs to return more than a small amount of data will return its data as a Recordset, or in dbGo terms, a TADODataSet.

To use `TADOConnection.OpenSchema`, you need an open a TADOConnection. The following example retrieves a list of primary keys for every table into a TADODataSet:

```
ADOConnection1.OpenSchema(siPrimaryKeys, EmptyParam, EmptyParam, ADODataSet1);
```

Each field in a primary key has a single row in the result set. So a table with a composite key of two fields has two rows. The two `EmptyParam` values indicate that these parameters are given empty values and are ignored.

When `EmptyParam` is passed as the second parameter, the result set includes all information of the requested type for the entire database. For many kinds of information, you will want to filter the result set. You can, of course, apply a traditional Delphi filter to the result set using the `Filter` and `Filtered` properties or the `OnFilterRecord` event. However, this applies the filter on the client side in this example. Using the second parameter, we can apply a more efficient filter at the source of the schema information. The filter is specified as an array of values. Each element of the array has a specific meaning relevant to the kind of data being returned. For example, the filter array for primary keys has three elements: the first is the catalog (*catalog* is ANSI-speak for the database), the second is the schema, and the third is the table name. This example returns a list of primary keys for the Customers table:

```
var
  Filter: OLEVariant;
begin
  Filter := VarArrayCreate([0, 2], varVariant);
  Filter[2] := 'CUSTOMERS';
  ADOConnection1.OpenSchema(
  siPrimaryKeys, Filter, EmptyParam, ADODataSet1);
end;
```

You can retrieve the same information using ADOX, and this warrants a brief comparison between `OpenSchema` and ADOX. ADOX is an additional ADO technology that allows you to retrieve and update schema information. It is ADO's equivalent to SQL's Data Definition Language (DDL, i.e., CREATE, ALTER, DROP) and Data Control Language (DCL, i.e., GRANT, REVOKE). ADOX is not directly supported in dbGo, but you can easily import the ADOX type library and use it successfully in Delphi applications. Unfortunately, ADOX is not as universally implemented as `OpenSchema`, so there are greater gaps. If you just want to retrieve information and not to update it, then `OpenSchema` is usually a better choice.

Using the Jet Engine

Now that you have some of the MDAC and ADO basics under your belt, we can take a moment out to look at the Jet engine. This engine is of great interest to some and of no interest to others. If you're interested in Access, Paradox, dBase, text, Excel, Lotus 1-2-3, or HTML, then this section is for you. If you have no interest in any of these formats, you can safely skip this section.

The Jet database engine is usually associated with Microsoft Access databases, and this is, indeed, its forte. However, the Jet engine is also an all-purpose desktop database engine, and this lesser-known attribute is where much of its strength lies. Since using the Jet engine with Access is its default mode and is straightforward, this section mostly covers use of non-Access formats, which are not so obvious.

Before we look at these formats, we should discuss the availability of the Jet engine. It was included with MDAC from v1.5c until, and including, v2.5. From MDAC v2.6, the Jet engine was dropped from MDAC (which accounts for the reduction in download size from 7.5 MB to 5.2 MB). You can download a distributable version of the Jet engine from `www.microsoft.com/data/download.htm`. Of course, if you have installed MDAC prior to v2.6, or Microsoft Access, Office, Excel, Visual Basic, or Visual C++ on your user's machine, then your user will already have the Jet engine.

There are two Jet OLE DB providers: the Jet 3.51 OLE DB Provider and the Jet 4.0 OLE DB Provider. The Jet 3.51 OLE DB Provider uses the Jet 3.51 engine and supports Access 97 databases only. If you intend to use Access 97 and not Access 2000, then you will get better performance using this OLE DB provider in most situations than using the Jet 4.0 OLE DB Provider. The Jet 4.0 OLE DB Provider supports Access 97, Access 2000, and Installable Indexed Sequential Access Method (IISAM) drivers. Installable ISAM drivers are those written specifically for the Jet engine to support access to ISAM formats such as Paradox, dBase, and text, and it is this facility that makes the Jet engine so useful and versatile. The complete list of ISAM drivers installed on your machine depends on what software you have installed on your machine. You can find this list by looking in the registry at

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM Formats
```

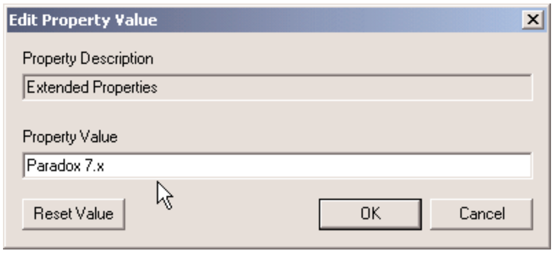
However, the Jet engine includes drivers for Paradox, dBase, Excel, text, and HTML.

Paradox

The Jet engine, naturally, expects to be used with Access databases. To use it with any database other than Access, you need to tell it which IISAM driver to use. This is a painless process that involves setting the Extended Properties connection string argument in the connection string editor. We'll do a quick example. Add a TADOTable to a form and invoke the connection string editor. Select the Jet 4.0 OLE DB Provider. Select the All page, locate the Extended Properties property, and double-click it to show the dialog box illustrated in Figure 16.3.

FIGURE 16.3:

Setting Extended Properties



Edit Property Value

Property Description: Extended Properties

Property Value: Paradox 7.x

Enter **Paradox 7.x** in the Property Value as shown and click OK. Now go back to the Connection tab and enter the name of the directory containing the Paradox tables. For example you can enter

```
c:\program files\common files\borland shared\data
```

which contains Delphi's DBDEMOS Paradox tables. Unfortunately, the Browse button showing the ellipses does not react to the Extended Properties value and always expects to select a file and not a directory, and so it has little value when used with Paradox databases. Click OK on both dialog boxes and select a table in the TADOTable's `TableName`. Set `Active` to True and you are now using Paradox through ADO.

Sadly, I have some bad news for Paradox users. Under certain circumstances, you will need to install the BDE in addition to the Jet engine. It is bizarre that in a chapter dedicated to ADO we are talking about the need to install the BDE in addition to MDAC, but depending on your application this may be true. Jet 4.0 requires the BDE in order to be able to update Paradox tables, but it doesn't require the BDE just to read them. The same is true for most releases of the Paradox ODBC Driver. As disastrous as this sounds, all is not lost. Microsoft has received justified criticism on this point and has made a new Paradox IISAM available that does not require the BDE. You can get these updated drivers from Microsoft Technical Support.

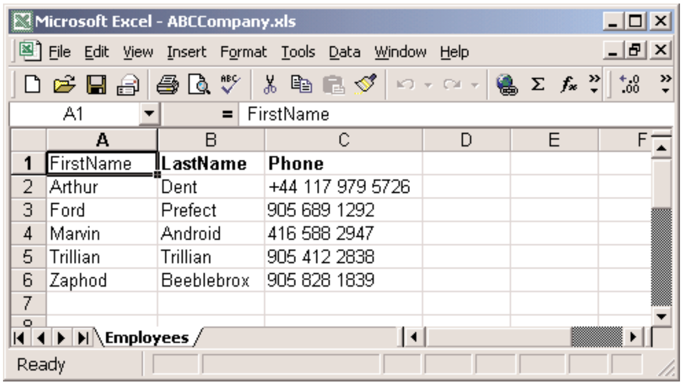
NOTE As you learn more and more about ADO, you will discover how much of ADO depends on the OLE DB provider and the DBMS (database management system) in question. You will see how the desktop databases such as Paradox and dBase have more restrictions and fewer functional features. If you are using Paradox simply because it is free, then you would be well advised to use another free database such as Access or MSDE. Alternatively, if leaving Paradox is not an option, then you should closely compare the BDE's support for Paradox with ADO's support for Paradox. In some cases, you will find the BDE's support better.

Excel

Excel is easily accessed using the Jet OLE DB Provider. Once again, we use the Extended Properties property and set it to Excel 8.0. Assume that we have an Excel spreadsheet called `ABCCompany.xls` that, in Excel, looks like Figure 16.4.

FIGURE 16.4:

ABCCompany.xls in Excel

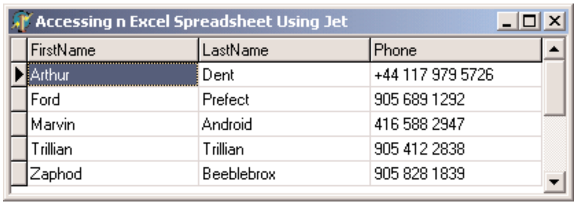

Microsoft Excel - ABCCompany.xls
File Edit View Insert Format Tools Data Window Help
A1
= FirstName
Employees
Ready

	A	B	C	D	E	F
1	FirstName	**LastName**	**Phone**			
2	Arthur	Dent	+44 117 979 5726			
3	Ford	Prefect	905 689 1292			
4	Marvin	Android	416 588 2947			
5	Trillian	Trillian	905 412 2838			
6	Zaphod	Beeblebrox	905 828 1839			
7						

Notice that the sheet is called Employees. Our mission is to open and read this file using Delphi. You can, of course, solve this problem by automating Excel with only a small knowledge of COM. However, the ADO solution is considerably easier to implement.

Ensure that your spreadsheet is not open in Excel. Add a TADODataSet to a form. Set `ConnectionString` to use the Jet 4.0 OLE DB Provider and set Extended Properties to Excel 8.0. In the Connection tab, set the database name to the full file and path specification of the Excel spreadsheet. Close the connection string editor. The TADODataSet component works by opening or executing a value in its `CommandText` property. This value might be the name of a table or an SQL statement or a stored procedure or the name of a file. You specify how this value is interpreted by setting the `CommandType` property. Set `CommandType` to cmdTableDirect to indicate that the value in `CommandText` is the name of a table and that all columns should be returned from this table. Select `CommandText` in the Object Inspector and you will see a drop-down arrow. Drop down the arrow and a single "table" will be displayed: Employees$. (Excel workbooks are suffixed with a $.) Set `Active` to True, add a TDataSource and a TDBGrid and connect them altogether, and you will see the Excel spreadsheet. It will be a little difficult to view in the grid because each column has a width of 255 characters. You can change this either by adding columns to the grid and changing their `Width` properties, or by adding persistent fields and changing their `Size` or `DisplayWidth` properties. After a little rearranging, you should see something like Figure 16.5.

FIGURE 16.5:
ABCCompany.xls in Delphi



FirstName	LastName	Phone
Arthur	Dent	+44 117 979 5726
Ford	Prefect	905 689 1292
Marvin	Android	416 588 2947
Trillian	Trillian	905 412 2838
Zaphod	Beeblebrox	905 828 1839

Now save your application. If you run it from the IDE, you will discover the first of the limitations of the Excel IISAM: the XLS file is opened exclusively. To run the application, you will first need to close the application that is open in the IDE and then run it from Windows Explorer. When you run the program, you will notice another limitation of this IISAM driver: you can add new rows and edit existing rows, but you cannot delete rows.

Incidentally, you could have used either TADOTable or TADOQuery, instead of TADODataSet, but you need to be aware of how ADO treats symbols in things like table names and field names. If you were to use a TADOTable and drop down the list of tables, you would see the Employees$ table as you would expect. Unfortunately if you attempt to open the table, you will receive an error. The same is true for `SELECT * FROM Employees$` in a `TADOQuery`. The problem lies with the dollar sign in the table name. If you use characters such as dollars, dots, or, more importantly, spaces in table names or field names, then you must enclose the name in square brackets (e.g., `[Employees$]`).

Text Files

One of the very useful IISAM drivers that comes with the Jet engine is the Text IISAM. This driver allows you to read and update text files of almost any structured format. We will start with a simple text file to get up and running and then cover the variations later. Assume we have a simple text file called `NightShift.TXT` that contains the following text:

```
CrewPerson ,HomeTown
Neo        ,Cincinnati
Trinity    ,London
Morpheus   ,Milan
```

Add a TADOTable to a form, set its `ConnectionString` to use the Jet 4.0 OLE DB Provider, and set Extended Properties to Text. The Text IISAM considers a directory a database, so you need to enter the directory that contains the `NightShift.TXT` file as the database name. Back in the Object Inspector, drop down the list of tables in the `TableName` property. The "database" consists of all of the text files in this directory. You will notice that the dot in the filename has been converted to a hash, as in `NightShift#TXT`. Set `Active` to True, add a TDataSource and a TDBGrid and connect them altogether, and you will see the contents of the text file in a grid.

If your computer's settings are such that the decimal separator is a comma instead of a period (so that 1,000.00 is displayed as 1.000,00), then you will need to either change your Regional Settings (Start ➢ Settings ➢ Control Panel ➢ Regional Settings ➢ Numbers) or take advantage of SCHEMA.INI, described shortly.

Of course, the grid indicates that the widths of the columns are 255 characters. You can change these just as we did in Excel by adding persistent fields or columns to the grid and then setting the relevant width property. Alternatively you can define the structure of the text file more specifically using SCHEMA.INI.

Text files come in all shapes and sizes. Often you do not need to worry about the format of a text file because the Text IISAM takes a peek at the first 25 rows to see whether it can determine the format for itself. It uses this information and some additional information in the Registry to decide how to interpret the file and how to behave. If you have a file that doesn't match a regular format the Text IISAM can determine, then you can provide this information in the shape of SCHEMA.INI. SCHEMA.INI is an INI file located in the same directory as the text files to which it refers. It contains *schema* information, also called metadata, about any or all of the text files in the same directory. Each text file is given its own section, identified by the name of the text file, such as [NightShift.TXT].

Thereafter you can specify the format of the file, the names, types, and sizes of columns, any special character sets to use, and any special column formats (e.g., date/time, currency). Let's assume that we change our NightShift.TXT file to the following format:

```
Neo        |Cincinnati
Trinity    |London
Morpheus   |Milan
```

In this example, the column names are not included in the text file and the delimiter is a vertical bar. An associated SCHEMA.INI file might look something like the following:

```
[NightShift.TXT]
Format=Delimited(|)
ColNameHeader=False
Col1=CrewPerson Char Width 10
Col2=HomeTown Char Width 30
```

Regardless of whether or not you use a SCHEMA.INI file, you will encounter two limitations of the Text IISAM: rows cannot be deleted, and rows cannot be edited.

Importing and Exporting

The Jet engine is particularly adept at importing and exporting data. The process of exporting data is the same for each export format and consists of executing a SELECT statement with a special syntax. Let's start with an example of exporting data from the Northwind

Access database to a Paradox table. You will need an active TADOConnection, called ADOConnection1 in our example, that uses the Jet 4.0 OLE DB Provider to open the Northwind.mdb Access database. The following code exports the Customers table to a Paradox Customers.db file:

```
ADOConnection1.Execute('SELECT * INTO Customers ' +
   'IN "C:\Temp" "Paradox 7.x;" FROM CUSTOMERS');
```

Let's look at the pieces of this SELECT statement. The INTO clause specifies the new table that will be created by the SELECT statement; this table must not already exist. The IN clause specifies the database to which the new table is added; in Paradox, this is a directory that already exists. The clause immediately following the database is the name of the IISAM driver to be used to perform the export. *You must include the trailing semicolon at the end of the driver name*. The FROM clause is a regular part of any SELECT statement.

All export statements follow these same basic clauses, but you will find that some IISAM drivers have differing interpretations of what a database is. I'll do another couple of examples to demonstrate the differences. Here, we export the same data to Excel:

```
ADOConnection1.Execute('SELECT * INTO Customers ' +
   'IN "Northwind.xls" "Excel 8.0;" FROM CUSTOMERS');
```

A new Excel file called Northwind.xls is created in the application's current directory. A workbook called Customers is added, containing all of the data of the Customers table in Northwind.mdb. You can also export data to Excel by automating Excel, but if you have ever done this you will know that this ADO solution is simpler by far.

This next example exports the same data to HTML:

```
ADOConnection1.Execute('SELECT * INTO [Customers.htm] ' +
   'IN "C:\Temp" "HTML Export;" FROM CUSTOMERS');
```

In this example, the database is the directory, as it was for Paradox but not for Excel. The table name must include the .htm extension and, therefore, it must be enclosed in square brackets. Notice that the name of the IISAM driver is "HTML Export", not just "HTML", because this driver can only be used for exporting to HTML.

The last IISAM driver we'll look at in this investigation of the Jet engine is the sister to HTML Export: HTML Import. Add a TADOTable to a form, set its ConnectionString to use the Jet 4.0 OLE DB Provider and Extended Properties to HTML Import. Set the database name to the name of the HTML file created by the export a few moments ago—that is, C:\Temp\Customers.htm. Close the connection string editors and set the TableName to Customers. Open the table and you have just imported the HTML file! Bear in mind, though, that the name of this IISAM driver is "HTML Import", not just "HTML". If you attempt to update the data in any way, you'll receive an error because this driver is intended for import only. Finally, if you create your own HTML files containing tables and want to open these tables using this driver, then remember that the name of the table is the value of the CAPTION tag of the HTML TABLE.

Cursor Locations and Cursor Types

There are two properties of ADO datasets that have a fundamental impact on your application and are inextricably linked with each other: `CursorLocation` and `CursorType`. The key to a successful application and to understanding your dataset's behavior and capabilities and the performance of your application lies in understanding these two properties.

The `CursorLocation`, of type `TCursorLocation`, allows you to specify what is in control of the retrieval and update of your data. You have two choices: client (`clUseClient`) or server (`clUseServer`). Your choice affects your dataset's functionality, performance, and scalability.

A client cursor is managed by the ADO Cursor Engine. This engine is an excellent example of an OLE DB service provider: it provides a service to other OLE DB providers. The ADO Cursor Engine manages the data from the client side of the application. All data in the result set is retrieved from the server to the client when the dataset is opened. Thereafter, the data is held in memory and updates and manipulation are managed by the ADO Cursor Engine. One benefit is that manipulation of the data, after the initial retrieval, is considerably faster. Furthermore, as the manipulation is performed in memory, the ADO Cursor Engine is more versatile than most server-side cursors and offers facilities that cannot be reproduced by server-side cursors. I'll examine these benefits later, as well as other technologies that depend on client-side cursors such as disconnected and persistent recordsets.

A server-side cursor is managed by the DBMS. In a client/server database such as SQL Server, Oracle, or InterBase, this means that the cursor is managed physically on the server. In a desktop database such as Access or Paradox, the "server" location is simply a logical location, as the database is running on the desktop. Server-side cursors are often faster to load than client-side cursors because not all of the data is transferred to the client when the dataset is opened. This also makes them more suitable for very large result sets where the client has insufficient memory to hold the entire result set in memory. Often you can determine what kinds of features will be available to you with each cursor location by thinking through how the cursor works. A good example of how features determine cursor type is locking, which I will cover in more detail later. To place a lock on a record requires a server-side cursor, because there must be a conversation between the application and the DBMS.

Another issue that will affect your choice of cursor location is scalability. Server-side cursors are managed by the DBMS; in a client/server database, this will be located on the server. As more and more users use your application, the load on the server increases with each server-side cursor. A greater workload on the server means that the DBMS becomes a bottleneck faster, so the application is less scalable. You can achieve better scalability by using client-side cursors. The initial hit on opening the cursor is often heavier, because all the data is transferred to the client, but the maintenance of the open cursor can be lower. As you can see, many conflicting issues are involved in choosing the correct cursor location for your datasets.

Your choice of cursor location directly affects your choice of cursor type. To all intents and purposes there are four cursor types, but I will digress for a moment to explain why there is one unused value. There is a cursor type that means "unspecified." Many values in ADO signify an unspecified value, and I will cover them all here and explain why you won't have much to do with them. They exist in Delphi because they exist in ADO. ADO was mostly designed for Visual Basic and C programmers. In these languages, you use the objects directly without any of the assistance that dbGo provides. As such, you can create and open *recordsets*, as they are called in ADO-speak, without having to specify every value for every property. The properties for which a value has not been specified have an unspecified value. However, in dbGo we use components. These components have constructors, and these constructors initialize the properties of the components. So from the moment you create a dbGo component, it will usually have a value for each and every property. The consequence is that we have little need for the unspecified values in many enumerated types.

Back to the cursor types. Cursor types, of type `TCursorType`, affect how your data is read and updated. There are four choices: forward-only, static, keyset, and dynamic. Before we get too involved in all of the permutations of cursor locations and cursor types, you should be aware that there is only one cursor type available for client-side cursors: the static cursor. All other cursor types are only available to server-side cursors. I'll return to the subject of cursor type availability after we have looked at the various cursor types, in increasing order of expensiveness.

The least expensive cursor type, and therefore the type with the best possible performance, is the forward-only cursor, which, as the name implies, will let you navigate forward. The cursor reads the number of records specified by `CacheSize` (default of 1) and each time it runs out of records, it reads another `CacheSize` set. Any attempt to navigate backward through the result set beyond the number of records in the cache will result in an error.

Knowing how a forward-only cursor works should help you to understand why they do not support bookmarks. A bookmark normally allows your dataset to navigate by jumping to a selected row. Because you can only travel forward through a forward-only cursor and any bookmark placed will be behind the current cursor position, bookmarks are not supported. As such a forward-only cursor is not suitable for use in the user interface where the user can control the direction through the result set. However, it is eminently suitable for batch operations and reports, because these situations start at the top of the result set and work progressively toward the end, then the result set is closed.

A static cursor works by reading the complete result set and providing a window of CacheSize records into the result set. As the complete result set has been retrieved by the server, you can navigate both forward and backward through the result set. However, in exchange for this facility, the data is static—that is, updates, insertions, and deletions made by other users cannot be seen because the cursor's data has already been read.

A keyset cursor is best understood by breaking *keyset* down into its two words *key* and *set*. Key, in this context, refers to an identifier for each row. Often this will be a primary key. A keyset cursor, therefore, is a set of keys. When the result set is opened, the complete list of keys for the result set is read. If, for example, the dataset was a query like SELECT * FROM CUSTOMERS, then the list of keys would be built from SELECT CUSTID FROM CUSTOMERS. This set of keys is held until the cursor is closed. When the application requests data, the OLE DB provider reads the rows using the keys in the set of keys. Consequently, the data is always up to date. If another user changes a row in the result set, then the changes will be seen when the data is reread. However, the set of keys, itself, is static; it is read only when the result set is first opened. So if another user adds new records, these additions will not be seen. Deleted records become inaccessible, and changes to primary keys (you don't let your users change primary keys, do you?) are also inaccessible.

The last, and most expensive, cursor type is dynamic. A dynamic cursor is almost identical to a keyset cursor. The sole difference is that the set of keys is reread when the application requests data that is not in the cache. As the default for TADODataSet.CacheSize is 1, this is very frequent. You can imagine the additional load this places on the DBMS and the network and why this is the most expensive cursor. However, the result set can see and respond to the additions and deletions made by other users.

Ask and Ye Shall Not Receive

Now that we know all about cursor locations and cursor types, a word of warning: not all combinations of cursor location and cursor type are possible. Usually, this is a limitation imposed by the DBMS and/or the OLE DB provider as a result of the functionality and architecture of the DBMS. For example, client cursors always force the cursor type to static. You can see this for yourself. Add a TADODataSet to a form, set its ConnectionString to any database, set ClientLocation to clUseCursor and CursorType to ctDynamic. Now set Active to True and keep your eye on the CursorType; it changes to ctStatic. We learn an important lesson from this example:

> What you ask for is not necessarily what you get.

Always check your properties after opening a dataset for what you think you "know" is true.

Each OLE DB provider will make different changes according to different requests and circumstances, but to give you a rough idea of what you can expect here are a few examples. The Jet 4.0 OLE DB Provider changes most cursor types to keyset. The SQL Server OLE DB Provider often changes keyset and static to dynamic. The Oracle OLE DB Provider changes all cursor types to forward-only. The ODBC OLE DB Provider makes various changes according to the ODBC driver in use.

RecordCount = –1

Armed with all of this knowledge about cursors, we can explain why ADO datasets sometimes return –1 for their `RecordCount`. A forward-only cursor cannot know how many records are in the result set until it reaches the end, so it returns –1 for the `RecordCount`. A static cursor always knows how many records are in the result set, because it reads the entire set when it is opened, so it returns the number of records in its result set. A keyset cursor also knows how many records are in the result set, because it has to retrieve a fixed set of keys when the result set is opened, so it also returns a useful value for `RecordCount`. A dynamic cursor does not reliably know how many records are in the result set, because it is regularly rereading the set of keys, so it returns –1. You could, of course, avoid using `RecordCount` altogether and execute `SELECT COUNT(*) FROM tablename`, but this will be an accurate reflection of the number of records in the database, which is not necessarily the same as the number of records in the dataset.

Client Indexes

One of the many benefits of client-side cursors is the ability to create local, or *client*, indexes. You can try this out for yourself. Assuming that you have an ADO client-side dataset for the Northwind's Customer table, which has a grid attached to it, set the dataset's `IndexFieldNames` property to CompanyName. Immediately the grid will show that the data is in CompanyName order. There is an important point to make here: In order to index the data, ADO did not have to reread the data from its source. The index was created from the data in memory. This means not only is the creation of the index just about as fast as it could possibly be, but the network and the DBMS are not overloaded with transferring the same data over and over in different orders. The `IndexFieldNames` property has more potential. Set it to `Country;CompanyName` and you will see the data ordered first by country and then, within country, in company name order. Now set `IndexFieldNames` to `CompanyName DESC`. Be sure to write DESC in capitals and not "desc" or "Desc". I'm sure you won't be surprised to see that the data is now sorted in descending order.

This simple but powerful feature allows us to solve one of the great bugbears of database developers. From time to time users seem to ask the inevitable, and quite reasonable, question, "Can I click the columns of the grid to sort my data?" There are three traditional answers to this question:

- "Yes. I can replace all of my grids with non–data-aware controls such as `TListView` that have the sorting built into the control. Of course, I lose all of the benefits of data-aware controls with this solution."
- "Yes. I can trap the TDBGrid's `OnTitleClick` event and rebuild the SQL SELECT statement to include an appropriate ORDER BY clause and then reissue the SELECT statement. Of course, this will mean requerying exactly the same data as I already have just to get it in a different order, but the user is always right."
- "No. I admit that this would be a cool feature, but the extra load on the DBMS and the network is antisocial to other users."

Sadly, none of the above are acceptable answers. Client indexes to the rescue! Add the following `OnTitleClick` event to the grid:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  if ADODataSet1.IndexFieldNames = Column.Field.FieldName then
    ADODataSet1.IndexFieldNames := Column.Field.FieldName + ' DESC'
  else
    ADODataSet1.IndexFieldNames := Column.Field.FieldName
end;
```

This simple event checks to see whether the current index is built on the same field as the column. If it is, then a new index is built on the column but in descending order. If not, then a new index is built on the column. When the user clicks the column for the first time, it is sorted in ascending order, and when it is clicked for the second time, it is sorted in descending order. You could extend this to allow the user to Ctrl-click several column titles to build up more complicated indexes. Of course, all of this can be achieved using TClientDataSet, but that solution is not as elegant for two reasons: descending indexes must be built from scratch (because TClientDataSet does not support the DESC keyword) and existing ascending indexes cannot be changed to descending indexes (they must be deleted and rebuilt).

Cloning

ADO is crammed full of features. You can argue that "feature-rich" can translate into "footprint-rich," but it also translates into more powerful and reliable applications. One such powerful feature is cloning. A cloned recordset is a new recordset that has all of the same properties as the original from which it is cloned. First, I'll explain how you can create and use a clone, and then I'll explain why they are so useful.

You can clone a recordset, or, in dbGo-speak, a dataset, using the `Clone` method. You can clone any ADO dataset, but we will use TADOTable in this example. Add a TADOTable to a form and set its `ConnectionString` to use any OLE DB provider that returns rectangular data (e.g., the Jet or SQL Server OLE DB providers). Set its `TableName` to any table and open the table. Add a TDataSource and a TDBGrid to allow you to view the table. Now add a second TADOTable and a button with the following code:

```
ADOTable2.Clone(ADOTable1);
```

This line clones `ADOTable1` and assigns the clone to `ADOTable2`. If you add another TDataSource and TDBGrid to show `ADOTable2`, you will see a second view of the data. The two datasets have their own record pointers and other status information, so the clone does not interfere with its original copy.

This behavior makes them ideal for *black box programming*. This term comes from an old story, in which you could ask the black box any question at all and it would guarantee to provide an answer. You couldn't see inside the black box and you didn't know how it worked, just that it did. The black box did what it was supposed to and no more. It is this last part that is so essential to programming: functions and procedures do *no more than they are supposed to*. Sometimes this is also referred to as having "zero side effects." So in this example, the `CountSelected` function attempts to count all of the rows where the `Selected` field is True:

```
function CountSelected(
ADODataSet: TCustomADODataSet): integer;
var
  Bookmark: TBookmark;
begin
  Result := 0;
  Bookmark := ADODataSet.GetBookmark;
  try
    ADODataSet.First;
    while not ADODataSet.EOF do
    begin
      if ADODataSet.FieldByName('Selected').AsBoolean then
        Inc(Result);
      ADODataSet.Next;
    end;
    ADODataSet.GoToBookmark(Bookmark);
  finally
    ADODataSet.FreeBookmark(Bookmark);
  end;
end;
```

At first sight, this function appears to be a black box function. It dutifully saves the current row position of the dataset using a bookmark and restores the position before returning, so that the net effect of moving the row position is zero. This is a standard approach to black box programming: if you change something in your routine that you are not supposed to change, then you must restore it again afterward.

Sadly, this routine is not black box. The movement of the row pointer will be observed by any data-aware controls, and the user interface will be updated by every row movement. A better solution to this problem is to use cloning:

```
function CountSelected(ADODataSet: TCustomADODataSet): integer;
var
  Clone: TADODataSet;
begin
  Result := 0;
  Clone := TADODataSet.Create(nil);
  try
    Clone.Clone(ADODataSet);
    Clone.First;
    while not Clone.EOF do
    begin
      if Clone.FieldByName('Selected').AsBoolean then
        Inc(Result);
      Clone.Next;
    end;
    Clone.Close;
  finally
    Clone.Free;
  end;
end;
```

The new clone will not interfere with its original in any way, making it ideal for black box programming. In particular, note that the closing of the clone does not close the original or other clones. In fact, not even the closing of the original closes its clones. I will use cloning later in this chapter, but before we close this subject, there are two points worth mentioning.

WARNING A recordset must support bookmarks in order to be cloned, so forward-only and dynamic cursors cannot be cloned. You can determine whether a recordset supports bookmarks using the `Supports` method (e.g., `ADOTable1.Supports([coBookMark])`).

TIP One of the useful side effects of clones is that the bookmarks created by one clone are usable by all other clones.

Transaction Processing

Transaction processing allows developers to group together individual updates to a database into a single logical unit of work. The benefit is that a database engine can be told to accept or reject the complete unit of work as a single entity. The facility is present in nearly all of today's DBMSs, because it's essential for maintaining the integrity of the database. The classic example used to illustrate the need for transaction processing is the transfer of money to and from a bank account. The movement of money from one account into another consists of two steps: the removal of money from one account and the addition of the same money into another account. The entire transaction must either wholly succeed or wholly fail in order for integrity to be maintained. If only one half of the transaction succeeds, the money will either have been created (by adding money to one account while failing to remove it from the other) or destroyed (by removing money from one account while failing to add it to the other). Database programming is full of examples that require transaction processing, and in this section we will see how ADO handles this subject.

ADO's transaction processing support is controlled using a TADOConnection. The following list summarizes the relevant methods:

TADOConnection Method	Description	TDatabase Equivalent
`BeginTrans`	Begins a transaction	`StartTransaction`
`CommitTrans`	Commits a transaction	`Commit`
`RollbackTrans`	Rolls back a transaction	`Rollback`

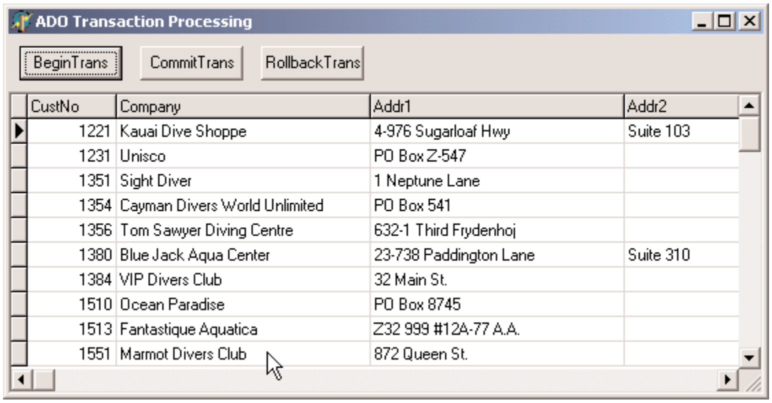
To investigate ADO's transaction processing support, we will build a simple test program. We will use this test program to investigate the different levels of transaction processing support offered by different OLE DB providers and different databases.

Create a new application and add a TADOConnection. Set the `ConnectionString` to use the ODBC OLE DB Provider and the DBDEMOS Paradox data supplied with Delphi. You might find it easier to first create an ODBC Data Source, say DBDEMOS, and refer to that. Set the `LoginPrompt` to False and `Connected` to True. Add a TADOTable, set `Connection` to ADOConnection1, `TableName` to Customers, and `Active` to True. Add a TDataSource and a TDBGrid and connect them so that the Customers table is shown in the grid. Add three buttons to the top of the form to execute each of the following commands:

```
ADOConnection1.BeginTrans;
ADOConnection1.CommitTrans;
ADOConnection1.RollbackTrans;
```

The running application should look something like Figure 16.6.

FIGURE 16.6:
ADO transaction processing



CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	

Click the BeginTrans button and you will receive an error indicating that the DBMS or the OLE DB provider is not capable of beginning a transaction. The problem lies with the Paradox ODBC Driver, which simply doesn't support transaction processing. You can find out what level of transaction processing support you have using the connection's Transaction DDL dynamic property—for example,

```
if ADOConnection1.Properties['Transaction DDL'].Value > DBPROPVAL_TC_NONE then
  ADOConnection1.BeginTrans;
```

The DBPROPVAL_TC_NONE constant comes from OLEDB.PAS along with several others like it. The related constants indicate that transaction support is available for Data Manipulation Language (DML) statements, meaning that SQL UPDATE, INSERT, and DELETE statements will all be included in transactions. ADO updates databases using SQL, so all of the dbGo components' DML methods (e.g., Append, Post, Delete) are also included in this list. The difference between the DBPROPVAL_TC constants is the level of support for Data Definition Language (DDL) commands included in a transaction. DDL is the set of SQL commands that alter the structure of the database and allow developers to perform actions such as adding and deleting columns and adding new tables. Given the nature of these commands, it is unlikely that you would want to mix DDL commands in a transaction with DML commands, so your interest in the Transaction DDL dynamic property might not extend beyond the DBPROPVAL_TC_NONE constant. If you do intend to include DDL commands in a transaction, you should check the other constants to see how the DDL will be treated by the DBMS. Some DBMSs will ignore the DDL, some will generate an exception if it is included in a transaction, some will cause certain DDL commands to lock a table, and others will automatically commit a transaction when the DDL is executed.

Let's get back to our test application. Change the TADOConnection's ConnectionString to use the Jet 4.0 OLE DB Provider and set its Extended Properties (in the All tab) to

Paradox 7.x. Set the database name to Delphi's DBDEMOS directory (`c:\program files\common files\borland shared\data`). Save the connection string and reopen the TADOTable. You should see no difference from the previous example. Now run the application and click the BeginTrans button. Success? Unfortunately, no. This is a cruel, cruel trick on the part of the Jet 4.0 OLE DB Provider. Make a few changes and click the RollbackTrans button; close the application down and restart it. You will see that the changes that you made were permanent and your act of rolling them back made no difference at all. As I said, it is a cruel trick, because transactions on a Paradox database cannot be rolled back, and this makes them rather useless. If you read through the section in this chapter on the Jet engine and, in particular, the section on Paradox, you will recall that I mentioned that the support for Paradox is lower than most developers would like. Transaction processing is one such example.

Now let's look at how Access handles transaction processing. Create an ODBC System Data Source Name using the ODBC Manager. Call it Northwind DSN, use the Microsoft Access ODBC driver, and set the database to `Northwind.mdb`. In Delphi, change the TADOConnection to use the ODBC OLE DB Provider and the new Northwind DSN. Now save the connection string, change the `TADOTable`'s `TableName` from Customer to Customers, and open it. Run the application and click the BeginTrans button. At last, transaction processing that works. You can make changes to the Northwind database and roll back the changes, and they will be rolled back! However, something you cannot do is start a transaction within a transaction. Try clicking the BeginTrans button twice and you get a "Cannot start more transactions on this session" error. This is a limitation of ODBC and not the Jet engine and is a useful example of why you should always try to locate an OLE DB provider for your DBMS instead of an ODBC driver.

Nested Transactions

If you try the same test again but use the Jet 4.0 OLE DB Provider, you will see that you can click the BeginTrans button five times before receiving an error on the sixth attempt. Jet supports *nested transactions*, which are transactions that exist within the context of another transaction. The nested or *inner* transaction can be committed or rolled back without affecting the outcome of the *outer* transaction. Let's work through a sequence of steps to be sure of how this works:

1. Begin a transaction.
2. Change the `ContactName` of the Around The Horn record from Thomas Hardy to Dick Solomon.
3. Begin a nested transaction.

4. Change the ContactName of the Bottom-Dollar Markets record from Elizabeth Lincoln to Sally Solomon.
5. Roll back the inner transaction.
6. Commit the outermost transaction.

The net effect is that only the change to the Around The Horn record is permanent. If, however, the inner transaction had been committed and the outer transaction rolled back, then the net effect would be that *none* of the changes were permanent (even the changes in the inner transaction). Although we are using Access to illustrate this behavior, the behavior is the same for all OLE DB providers that support nested transactions. This, of course, leads us to an ongoing theme in our ADO exploration: The ADO documentation simply states how any given feature is supposed to work when it is fully implemented by the OLE DB provider and the DBMS, but it does not necessarily follow that it *is* implemented for all OLE DB providers and all DBMSs. In the example of nested transactions, we have seen that ODBC does not support them and that the Jet OLE DB Provider supports up to five levels of nested transactions. The SQL Server OLE DB Provider does not support nesting.
In SQL Server 7.0 and 2000, full nested transaction support
is provided. In SQL Server 6.5, only "fake nesting" is supported, where all inner-transaction instructions are ignored. You can commit and roll back inner transactions without any effect. It is only the outermost transaction that decides whether the complete sum of all of the work is committed or rolled back.

There is another issue that you should consider if you intend to use nested transactions. TADOConnection has a property called Attributes, which determines how the connection should behave when a transaction is committed or rolled back. It is a set of TXActAttributes that, by default, is empty. There are only two values in TXActAttributes: xaCommitRetaining and xaAbortRetaining (this value is often mistakenly written as xaRollbackRetaining because this would have been a more logical name for it). When xaCommitRetaining is included in Attributes and a transaction is committed, a new transaction is automatically started. When xaAbortRetaining is included in Attributes and a transaction is rolled back, a new transaction is automatically started. This means that if you include these values in Attributes, a transaction will always be in progress, because when you end one transaction another will always be started. Most programmers prefer to be in greater control of their transactions than allowing them to be automatically started, so these values are not commonly used. However, they have a special relevance to nested transactions. If you nest a transaction and set `Attributes` to `[xaCommitRetaining, xaAbortRetaining]`, then the outermost transaction can never be ended. Consider the sequence of events:

1. An outer transaction is started.
2. An inner transaction is started.
3. The inner transaction is committed or rolled back.
4. A new inner transaction is automatically started as a consequence of the `Attributes` property.

The outermost transaction can never be ended because a new inner transaction will also be started when one ends. The conclusion is that the use of the `Attributes` property and the use of nested transactions should be considered mutually exclusive.

Lock Types

ADO supports four different approaches to locking your data for update. In this section I will provide an overview of the four approaches, and in subsequent sections we will take a closer look. The four approaches are made available to you through the dataset's `LockType` property, of type `TLockType`, and can be ltReadOnly, ltPessimistic, ltOptimistic, or ltBatchOptimistic (there is, of course, an ltUnspecified but, for the reasons mentioned earlier, we are ignoring "unspecified" values).

The ltReadOnly value specifies that the data is read-only and cannot be updated. As such, there is effectively no locking control required because the data cannot be updated.

The ltPessimistic and ltOptimistic values offer the same "pessimistic" and "optimistic" locking control that the BDE offers. One important benefit that ADO offers over the BDE in this respect is that the choice of locking control remains yours. If you use the BDE, the decision to use pessimistic or optimistic locking is made for you by the BDE driver you use. If you use a desktop database such as dBase or Paradox, then the BDE driver uses pessimistic locking; if you use a client/server database such as InterBase, SQL Server, or Oracle, the BDE driver uses optimistic locking.

Pessimistic Locking

The words *pessimistic* and *optimistic* in this context refer to the developer's expectations of conflict between user updates. Pessimistic locking assumes that there is a high probability that users will attempt to update the same records at the same time and that a conflict is likely. In order to prevent such a conflict, the record is locked when the edit begins. The record lock is maintained until the update is completed or cancelled. A second user who attempts to edit the same record at the same time will fail in their attempt to place their record lock and will receive a “Could not update; currently locked” exception.

This approach to locking will be familiar to developers who have worked with desktop databases such as dBase and Paradox. The benefit is that the user knows that if they can begin editing a record, then they will succeed in saving their update. The disadvantage of pessimistic locking is that the user is in control of when the lock is placed and when it is removed. If the user is skilled with the application, then this could be as short as a couple of seconds. However, in database terms a couple of seconds is an eternity. At the worst end of the scale, the user can begin an edit and go to lunch, and the record could be locked until the user returns. As a consequence of this, most proponents of pessimistic locking guard against this eventuality by using a TTimer or other such device to time out any locks after a certain amount of keyboard and mouse inactivity.

Another problem with pessimistic locking is that it requires a server-side cursor. Earlier we looked at cursor locations and saw that they have an impact on the availability of the different cursor types. Now we can see that cursor locations also have an impact on locking types. Later in this chapter we will see more benefits of client-side cursors, and if you choose to take advantage of these benefits, then you’ll be unable to use pessimistic locking.

Pessimistic locking is one of the areas of ADOExpress/dbGo that changed in Delphi 6. This section describes the way pessimistic locking works in version 6 (which is now the same as for ADO). Create a new application and add a TADODataSet. Set its `ConnectionString` to use either the Jet or SQL Server Northwind database. Set its `CommandType` to cmdTable and `CommandText` to Customers. As we will be using pessimistic locking, we must set the `CursorLocation` to clUseServer and `LockType` to ltPessimistic. Finally, set `Active` to True. Add a TDataSource and a TDBGrid, connect them altogether, and ensure that the grid is aligned to client.

Now for the test. Run the application and begin editing a record. Using Windows Explorer, run a second copy of the same application and attempt to edit the same record; you will fail because the record is locked by another user.

If you were using ADOExpress in Delphi 5, the attempt to edit the same record at the same time would have succeeded, because ADOExpress in version 5 did not lock the record at the beginning of the edit. The work-around for this problem in Delphi 5 involved creating a clone of the original recordset and forcing a record lock in the clone before an edit, and releasing the record lock before the actual post or when the edit was cancelled.

Jet Page and Row Locking

This section demystifies some of the issues surrounding locking in Microsoft's Jet engine. If you have no interest in Access as a database, you can safely skip over this section.

The Jet 4.0 OLE DB Provider, which you will recall is used primarily for Access 2000 databases although it can also be used with Access 97 databases, supports both page-level and row-level locking. The Jet 3.5 OLE DB Provider, which is used solely for Access 97 databases, supports page-level locking only. A *page* is a length of data. The page in question completely contains the record for which the lock is required. In Jet 3.5, a page is 2 KB in length, and in Jet 4.0 it is 4 KB. It is unlikely that a record is exactly 2 KB or 4 KB, so the locking of a single record usually includes the locking of one or more subsequent records. Clearly this locks additional records unnecessarily, which is a disadvantage of page locking and is the main reason why Jet 4.0 offers a choice of page or row locking.

Row locking, sometimes referred to as Alcatraz in the Jet engine, allows a single row to be locked individually with no additional space locked. As such, this solution provides the least lock contention. The path to understanding locking in Jet lies in Jet's dynamic properties. Most of your control over Jet locking is provided by the ADO Connection object's dynamic properties. The first and most important dynamic property is Jet OLEDB:Database Locking Mode, which can be revealed in a TADOConnection as follows:

```
ShowMessage(ADOConnection1.Properties[
  'Jet OLEDB:Database Locking Mode'].Value);
```

By default this value is 1, which means that the connection will allow recordsets to choose between row locking and page locking. The only alternative value is 0, which forces page locking. However, in order to ensure that all users use the same locking mechanism, the first user who opens the database dictates the locking mode used by all users. This mode remains in force until all users have disconnected from the database.

The second part of the locking jigsaw is the recordset. The recordset itself can specify its locking mode, using the Jet OLEDB:Locking Granularity dynamic property. By default this value is 2, which indicates that it should use row-level locking. Setting this value to 1 indicates that the recordset should use page locking. The Jet OLEDB:Locking Granularity dynamic property is ignored if the connection's Jet OLEDB:Database Locking Mode is not 1. You will have to use the `OnRecordsetCreate` event added in Delphi 6 to set the dynamic property, because it can only be set when the recordset is created but closed. (`OnRecordsetCreate` was not yet functional as this book went to press. Also, if you are using ADOExpress in Delphi 5, you will have to modify the source code of `TCustomADODataSet.OpenCursor` to add in your own `OnRecordsetCreate` event immediately after the recordset is created.) Thus, so far, Jet 4.0 uses row-level locking by default. This is true but there is an extra twist to add to this tale.

Row-level locking is only the default for recordsets. It is not the default for SQL you execute directly, or what BDE developers refer to as “non-passthrough SQL.” For SQL statements you write yourself and execute directly, the default locking mode is still page locking, even using Jet 4.0.

Having gone through these trials and tribulations you will be pleased to learn that Jet provides a significant level of control over its locking facilities. Once again, this control is offered by dynamic properties and is available through TADOConnection. The following list shows the relevant dynamic properties:

Property	Description
Jet OLEDB:Lock Delay	Milliseconds to wait before attempting to reacquire a lock (default is 0)
Jet OLEDB:Lock Retry	Number of times to retry a failed lock (default is 0)
Jet OLEDB:Max Locks Per File	Maximum number of locks that Jet can place on a database (default is 9500)
Jet OLEDB:Page Locks To Table	Number of page locks before Jet promotes to a table lock (default is 0)

Updating JOINs

One of the reasons why people used to turn to cached updates in the BDE and, more recently, TClientDataSet, is to make an SQL JOIN updatable. Consider the following SQL equi-join:

```
SELECT * FROM Products, Suppliers
WHERE Products.SupplierID=Suppliers.SupplierID
```

This statement provides a list of products and the details of the suppliers of those products. The BDE considers any SQL JOIN to be read-only because inserting, updating, and deleting rows in a join is ambiguous. For example, should the insert of a row into the above join result in a new product and also a new supplier or just a new product?

The BDE supports cached updates, which allow the developer to resolve this ambiguity by specifying exactly what the developer wants to happen. Although the BDE’s cached updates implementation is often flawed, and cached updates are now discouraged in favor of the more reliable TClientDataSet, the concept is sound.

ADO supports an equivalent to cached updates, called batch updates, which are very similar. In the next section we will take a closer look at ADO’s batch updates, what they can offer you, and why they are so important. However, in this section they will not be needed to solve the problem of updating a join for a very simple reason: in ADO, joins are naturally updatable. Place a TADOQuery on a form and set its connection string to use a Northwind database. Enter the SQL join above in its SQL property and set `Active` to True. Add a TDataSource and a TDBGrid and connect them altogether and run the program. Now edit one of the Product's fields and save the changes (by moving off the record). No error occurs because the update has been applied successfully. ADO has taken a more practical approach to the problem: it has made some intelligent guesses. In an ADO join, each field object knows which underlying table it belongs to. If you update a field of the Products table and post the change, then a SQL UPDATE statement is generated to update the field in the Products table. If you change a field in the Products table and a field in the Suppliers table, then two SQL UPDATE statements are generated, one for each table.

The inserting of a row into a join follows a similar behavior. If you insert a row and enter values for the Products table only, then a SQL INSERT statement is generated for the Products table. If you enter values for both tables, two SQL INSERT statements are generated, one for each table. The order in which the statements are executed is important, because the new product might relate to the new supplier, so the new supplier is inserted first.

The biggest problem with ADO's solution can be seen when a row in a join is deleted. The deletion attempt will appear to fail. The exact message you see depends on the version of ADO you are using and the DBMS, but it will be along the lines that you cannot delete the row because other records relate to it. The error message can be confusing. In our scenario, the error message implies that a product cannot be deleted because there are records that relate to the product. The error occurs whether the product has any related records or not. The explanation can be found by following through the same logic for deletions as for insertions. Two SQL DELETE statements are generated: one for the Suppliers table and then another for the Products table. Contrary to appearances, the DELETE statement for the Product table succeeds. It is the DELETE statement for the Suppliers table that fails, because the Supplier cannot be deleted while it still has dependent records.

If you are curious about the SQL statements that get generated and you use SQL Server, you can see these statements using SQL Server Profiler.

Despite understanding how this works, a better way of looking at this problem is through the user's eyes. From their point of view, when they delete a row in the grid, do they intend to delete just the product or both the product and the supplier? I would wager that 99% of users expect the former and not the latter. Fortunately you can achieve exactly this with our old friend, the dynamic property—in this case, the Unique Table dynamic property. You can specify that deletes refer to just the Products table and not to Suppliers using the following line of code:

```
ADOQuery1.Properties['Unique Table'].Value := 'Products';
```

As this value cannot be assigned at design time, the next best alternative is to place this line in the form's OnCreate event.

For me, updatable joins are just one of many examples of how the designers of ADO have replaced traditional problems with elegant solutions.

Batch Updates

Batch updates are ADO's equivalent to the BDE's cached updates; they are similar in functionality, syntax, and, to some extent, implementation, with the all-important difference being that their implementation is not fundamentally flawed. The idea is the same for both database technologies: By using batch/cached updates, any changes you make to your records can be made in memory and then later the entire "batch" of changes can be submitted as one operation. There are some performance benefits to this approach, but there are more practical reasons why this technology is a necessity: the user might not be connected to the database at the time they make their updates. This would be the case in the infamous "briefcase" application, which we will return to later, but this can also be the case in Web applications that use another ADO technology, Remote Data Services (RDS).

You can enable batch updates in any ADO dataset by setting LockType to ltBatchOptimistic before the dataset is opened. In addition, you will need to set the CursorLocation to clUseClient, as batch updates are managed by ADO's cursor engine. Hereafter, changes are all made to a "delta" (i.e., a list of changes). The dataset looks to all intents and purposes as if the data has changed, but the changes have only been made in memory; they have not been applied to the database. To make the changes permanent, use UpdateBatch (equivalent to cached updates' ApplyUpdates):

```
ADODataSet1.UpdateBatch;
```

(Fortunately, there is no equivalent to the cached update's CommitUpdates method, because the successful changes are automatically removed from the batch.) To reject the entire batch of updates, use either CancelBatch or CancelUpdates. There are many similarities in method and property names between ADO's batch updates and BDE's cached updates and TClientDataSet. UpdateStatus, for example, can be used in exactly the same way as for cached updates to identify records according to whether they have been inserted, updated, deleted, or unmodified. This is particularly useful for highlighting records in different colors in a grid or showing their status on a status bar. Some differences between the syntaxes are slight, such as changing RevertRecord to CancelBatch(arCurrent). Others require more effort.

One useful cached update feature that is not present in ADO batch updates is the dataset's UpdatesPending property. This property is true if changes have been made but not yet applied. This is particularly useful in a form's OnCloseQuery event:

```
procedure TForm1.FormCloseQuery(
Sender: TObject; var CanClose: Boolean);
begin
  CanClose := True;
  if ADODataSet1.UpdatesPending then
    CanClose := (MessageDlg('Updates are still pending' #13 +
      'Close anyway?', mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

However, with a little knowledge and a little ingenuity we can implement a suitable ADOUpdatesPending function. The little knowledge is that ADO datasets have a property called FilterGroup, which is a kind of filter. Unlike a dataset's Filter property, which filters the data based on a comparison of the data against a condition, FilterGroup filters based on the status of the record. One such status is fgPendingRecords, which includes all records that have been modified but not yet applied. So to allow the user to look through all of the changes they have made so far, you need only execute two lines:

```
ADODataSet1.FilterGroup := fgPendingRecords;
ADODataSet1.Filtered := True;
```

Naturally, the result set will now include the records that have been deleted. If you try this yourself, the effect that you will see will depend on the version of dbGo you have and the patches you have applied to it. In early versions of ADOExpress, the deleted record showed the fields of the previous record. This either was confusing (if there was a previous record) or resulted in a fatal error (if there wasn't). In later versions, the fields are just left blank, which also is not very helpful because you don't know what record has been deleted.

Back to the UpdatesPending problem. The "little ingenuity" is the knowledge of clones, discussed earlier. The idea of the ADOUpdatesPending function is that it will set the FilterGroup to restrict the dataset to only those changes that have not yet been applied. All we need to do is to see whether there are any records in the dataset once the FilterGroup has been applied. If there are, then some updates are pending. However, if we do this with the actual dataset, then the setting of the FilterGroup will move the record pointer and the user interface will be updated. The best solution is to use a clone.

```
function ADOUpdatesPending(ADODataSet: TCustomADODataSet): boolean;
var
  Clone: TADODataSet;
begin
  Clone := TADODataSet.Create(nil);
  try
    Clone.Clone(ADODataSet);
```

```
      Clone.FilterGroup := fgPendingRecords;
      Clone.Filtered    := True;
      Result := not (Clone.BOF and Clone.EOF);
      Clone.Close;
    finally
      Clone.Free;
    end;
  end;
```

In this function we clone the original dataset, set the FilterGroup, and check to see whether the dataset is at both beginning of file and also end of file. If it is, then no records are pending.

Optimistic Locking

Earlier we looked at the LockType property and saw how pessimistic locking worked. In this section, we'll look at optimistic locking, not only because it is the preferred locking type for medium- to high-throughput transactions but also because it is the locking scheme employed by batch updates.

Optimistic locking assumes that there is a low probability that users will attempt to update the same records at the same time and that a conflict is unlikely. As such, the attitude is that all users can edit any record at any time, and we deal with the consequences of conflicts between different users' updates to the same records when the changes are saved. Thus, conflicts are considered an exception to the rule. This means that there are no controls to prevent two users from editing the same record at the same time. The first user to save their changes will succeed. The second user's attempt to update the same record might fail. This behavior is essential for briefcase applications and Web applications, where there is no permanent connection to the database and, therefore, no way to implement pessimistic locking. In contrast with pessimistic locking, optimistic locking has the additional considerable benefit that resources are consumed only momentarily and, therefore, the average resource usage is much lower, making the database more scalable.

Let's consider an example. Assume that we have a TADODataSet connected to the Customers table of Northwind, that LockType is set to ltBatchOptimistic, and the contents are displayed in a grid. Assume that we also have a button to call UpdateBatch. Run the program twice and begin editing a record in the first copy of the program. Although for the sake of simplicity we will be demonstrating a conflict using just a single machine, the scenario and subsequent events are unchanged when using multiple machines. In this example I will choose the Bottom-Dollar Markets company in Canada and change the name to Bottom-Franc Markets. Save the change, move off the record to post it, and click the button to update the batch. Now, in the second copy of the program, locate the same record and change the company name to Bottom-Pound Markets. Move off the record and click the button to update the batch. It will fail. As with many ADO error messages, the exact message you receive will depend not only on the version of ADO you are using but also on how closely you followed the example. In ADO 2.6, the error message is “Row cannot be located for updating. Some values may have been changed since it was last read.” This is the nature of optimistic locking. The update to the record is performed by executing the following SQL statement:

```
UPDATE CUSTOMERS SET CompanyName="Bottom-Pound Markets"
WHERE CustomerID="BOTTM" AND CompanyName="Bottom-Dollar Markets"
```

The number of records affected by this update statement is expected to be 1, because it locates the original record using the primary key and the contents of the `CompanyName` field as it was when the record was first read. In our example, however, the number of records affected by the UPDATE statement is 0. This can only occur if the record has been deleted, or the record’s primary key has changed, or the field that we are changing was changed by someone else. Hence, the update fails.

If our “second user” had changed the `ContactName` field and not the `CompanyName` field, then the UPDATE statement would have looked like this:

```
UPDATE CUSTOMERS SET ContactName="Liz Lincoln"
WHERE CustomerID="BOTTM" AND ContactName="Elizabeth Lincoln"
```

In our scenario, this statement would have succeeded because the other user didn’t change the primary key or the contact name.

This behavior differs from the BDE’s behavior in the same scenario. In this example, using the BDE the attempt to update the contact name would have failed because, by default, the BDE includes every field in the WHERE clause. The consequence of this is that any change to the record will fail if any other user has already changed any field, regardless of whether the changed fields are the same fields or different fields. Fortunately, both the BDE and ADO allow you to specify how you want to locate the original record: in the BDE you use the `UpdateMode` property, and in ADO the Update Criteria dynamic property of a dataset. The following list shows the possible values that can be assigned to this dynamic property:

Constant	**Locate Records By**
adCriteriaKey	Primary key columns only
adCriteriaAllCols	All columns
adCriteriaUpdCols	Primary key columns and changed columns only
adCriteriaTimeStamp	Primary key columns and a timestamp column only

The reason why the BDE and ADO differ in their behavior is that their defaults differ. The BDE’s default behavior is equivalent to ADO’s adCriteriaAllCols, whereas ADO’s default is adCriteriaUpdCols. Don’t fall into the trap of thinking that one of these settings is better than another for your whole application. In practice, your choice of setting will be influenced by the contents of each table. Say that the Customers table has just CustomerID, Name, and City fields. In this case, the update of any one of these fields is logically not mutually exclusive with the update of any of the other fields, so a good choice for this table would be adCriteriaUpdCols (i.e., the default). If, however, the Customers table included a PostalCode field, then the update of a PostalCode field would be mutually exclusive with the update of the City field by another user (because if the city changes, then surely so should the postal code, and possibly vice versa). In this case, you could argue that adCriteriaAllCols would be a safer solution.

Another issue to be aware of is how ADO deals with errors during the update of multiple records. Using the BDE's cached updates and TClientDataSet, you can use the OnUpdateError event to handle each update error as the error occurs and resolve the problem before moving on to the next record. In ADO, you cannot establish such a dialog. You can monitor the progress and success or failure of the updating of the batch using the dataset's OnWillChangeRecord and OnRecordChangeComplete, but you cannot revise the record and resubmit it during this process as you can with the BDE and TClientDataSet. There's more: if an error occurs during the update process, the updating does not stop. It continues to the end until all updates have been applied or have failed. This can produce a rather unhelpful and blatantly incorrect error message. If more than one record cannot be updated, or the single record that failed is not the last record to be applied, then the error message in ADO 2.6 is "Multiple-step OLE DB operation generated errors. Check each OLE DB status value, if available. No work was done." The last sentence is the problem; it states that "No work was done," but this is incorrect. It is true that no work was done on the record that failed, but other records were successfully applied and their updates stand.

Resolving Update Conflicts

As a consequence of the nature of applying updates, the approach that you need to take to update the batch is to update the batch, let the individual records fail, and then deal with the failed records once the process is over. You can determine which records have failed by setting the dataset's FilterGroup to fgConflictingRecords:

```
ADODataSet1.FilterGroup := fgConflictingRecords;
ADODataSet1.Filtered := True;
ADODataset1.Recordset.Resync(adAffectGroup, adResyncUnderlyingValues);
```

For each failed record, you can inform the user of three critical pieces of information about each field using the following TField properties:

Property	Description
NewValue	The value this user changed it to
CurValue	The new value from the database
OldValue	The value when first read from the database

Users of TClientDataSet will be aware of the very handy `TReconcileErrorForm` dialog, which wraps up the process of showing the user the old and new records and allows them to specify what action to take. Unfortunately, there is no ADO equivalent to this form, and `TReconcileErrorForm` has been written with TClientDataSet so much in mind that it is difficult to convert it for use with ADO datasets.

One last gotcha to point out when using these `TField` properties: They are taken straight from the underlying ADO Field objects to which they refer. This means, as is common in ADO, that you are at the mercy of your chosen OLE DB provider to support the features you hope to use. All is well for most providers, but the Jet OLE DB Provider returns the same value for `CurValue` as it does for `OldValue`. In other words, if you use Jet, you cannot determine what the other user changed the field to unless you resort to your own measures.

Disconnected Recordsets

All this knowledge of batch updates allows us to take advantage of our next ADO feature: disconnected recordsets. A disconnected recordset is a recordset that has been disconnected from its connection. What is impressive about this feature is that the user cannot tell the difference between a regular recordset and a disconnected one; their feature sets and behavior are almost identical. To disconnect a recordset from its connection, the `CursorType` must be set to clUseClient and the `LockType` must be set to ltBatchOptimistic. You then simply tell the dataset that it no longer has a connection:

```
ADODataSet1.Connection := nil;
```

Hereafter, the recordset will continue to contain the same data, support the same navigational features, and allow records to be added, edited, and deleted. The only relevant difference is that you cannot update the batch because you need to be connected to the server to update the server. To reconnect the connection (and use `UpdateBatch`):

```
ADODataSet1.Connection := ADOConnection1;
```

This same feature is available to the BDE and other database technologies by switching over to TClientDataSets, but the beauty of the ADO solution is that you can build your entire application using dbGo dataset components and be unaware of disconnected recordsets. At the point that you discover this feature and want to take advantage of it, you can continue to use the same components that you always used.

So why would you want to disconnect your recordsets? For two reasons:

- To keep the total number of connections lower
- To create a briefcase application

I'll cover keeping down the number of connections here and return to briefcase applications later. Most regular client/server business applications open tables and maintain a permanent connection to their database while the table is open. However, there are usually only two reasons why you want to be connected to the database: to retrieve data and to update data. If you change your regular client/server application so that, after the table is opened and the data retrieved, then the dataset is disconnected from the connection and the connection dropped, your user will be none the wiser and the application will not need to maintain an open database connection. The following code shows the two steps:

```
ADODataSet1.Connection := nil;
ADOConnection1.Connected := False;
```

The only other point at which a connection is required is when the batch of updates needs to be applied, so the update code would look like this:

```
ADOConnection1.Connected := True;
ADODataSet1.Connection := ADOConnection1;
try
  ADODataSet1.UpdateBatch;
finally
  ADODataSet1.Connection := nil;
  ADOConnection1.Connected := False;
end;
```

If this approach were followed throughout the application, the average number of open connections at any one time would be minimal because the connections would only be open for the small amount of time that they are required. The consequence of this change is scalability; The application will be able to cope with significantly more simultaneous users than one that maintains an open connection. The downside, of course, is that the reopening of the connection can be a lengthy process on some, but not all, database engines, so the application will be slower to update the batch.

Connection Pooling

All of this talk of dropping connections and reopening them brings us to the subject of connection pooling. Connection pooling, not to be confused with session pooling, allows connections to a database to be reused once they have been finished with. This happens automatically and, if your OLE DB provider supports it and it is enabled, you need take no action to take advantage of connection pooling. There is a single reason why you would want to pool your connections: performance. The problem with database connections is that it can take time to establish a connection. In a desktop database such as Access, this is typically a small amount of time. In a client/server database such as Oracle used on a network, this time could be measured in seconds. Given such an expensive (in performance terms) resource, it makes sense to promote its reuse. With ADO's connection pooling enabled, ADO Connection objects are placed in a pool when the application "destroys" them. Subsequent attempts to create an ADO connection will automatically search the connection pool for a connection with the same connection string. If a suitable connection is found, it is reused; otherwise, a new connection is created. The connections themselves stay in the pool until they are reused, the application closes, or they time out. By default, connections will time out after 60 seconds, but from MDAC 2.5 onward you can set this using the `HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>\SPTimeout` registry key. The connection pooling process occurs seamlessly, without the intervention or knowledge of the developer. This process is similar to the BDE's database pooling under Microsoft Transaction Server (MTS) and COM+, with the important exception that ADO performs its own connection pooling without the aid of MTS or COM+.

By default, connection pooling is enabled on all of the MDAC OLE DB providers for relational databases (including SQL Server and Oracle) with the notable exception of the Jet OLE DB Provider. If you use ODBC you should choose between ODBC's connection pooling and ADO's connection pooling, but you should not use both. From MDAC 2.1 on, ADO's connection pooling is enabled and ODBC's is disabled.

NOTE Connection pooling does not occur on Windows 95 regardless of the OLE DB provider.

To be truly comfortable with connection pooling, you will need to see the connections getting pooled and timed out. Unfortunately, there are no adequate ADO connection pool spying tools available at the time of writing, so we will use SQL Server's Performance Monitor as it can accurately spy on SQL Server database connections. Figure 16.7 is a look at SQL Server's Performance Monitor with all of the "counters" deleted except User Connections. This allows us to concentrate of the subject of connection pooling.

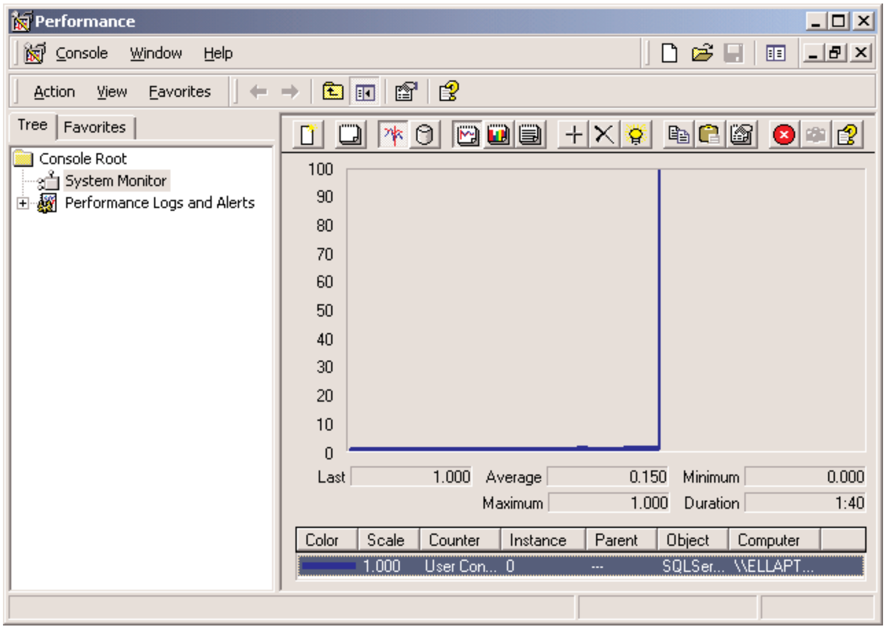
The Last field under the graph shows us the number of active connections to the database.

To see how connection pooling works, you can set up a very simple test. Create a new application and add a TADOConnection to the form. Set the `ConnectionString` to use the SQL Server OLE DB Provider and the Northwind database but leave `Connected` as False. Now add a check box with the following `OnClick` event:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  ADOConnection1.Connected := CheckBox1.Checked;
end;
```

FIGURE 16.7:

SQL Server's Performance Monitor



Last 1.000 Average 0.150 Minimum 0.000
Maximum 1.000 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
▬	1.000	User Con...	0	---	SQLSer...	\\ELLAPT...

Run the program and make sure that you can see the Performance Monitor at the same time. Now click the check box to open the connection. In the Performance Monitor, you will see the connection count increase by one. Now close the application and the count immediately decreases by one, because the connection pool is destroyed with the application. Now rerun the program, check the check box, and check it a second time to close the connection. You will see that the connection count does not decrease by one. Observe Performance Monitor for a further 60 seconds, and the connection count will then decrease by one when the pooled connection times out.

You can enable or disable connection pooling either in the Registry or in the connection string. The key in the Registry is `OLEDB_SERVICES` and can be found at `HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>`. It is a bit array that allows you to disable several OLE DB services, including connection pooling, transaction enlistment, and the cursor engine. To disable connection pooling using the connection string, include `";OLE DB Services=-2"` at the end of the connection string. To enable connection pooling for the Jet OLE DB Provider, you can include `";OLE DB Services=-1"` at the end of the connection string, which enables all OLE DB services.

Persistent Recordsets

One of the very useful features that contributes to the briefcase model is persistent recordsets. These allow you to save the contents of any recordset to a local file, which can be loaded later. Apart from aiding with the briefcase model, this feature allows developers to create true single-tier applications. It means that you can deploy a database application without having to deploy a database. This makes for a very small footprint on your client's machine.

You can "persist" your datasets using the SaveToFile method:

```
ADODataSet1.SaveToFile('Local.ADTG');
```

This will save the data and its delta in a file on your hard disk. You can reload this file using the LoadFromFile method, which accepts a single parameter indicating the file to load. The format of the file is Advanced Data Table Gram (ADTG), which is a proprietary Microsoft format. It does, however, have the advantage of being very efficient. If you prefer, you can save the file as XML by passing a second parameter to SaveToFile:

```
ADODataSet1.SaveToFile('Local.XML', pfXML);
```

However, ADO does not have its own built-in XML parser (as TClientDataSet does), so it must use the MSXML parser. Your user must either install Internet Explorer 5 or later or download the MSXML parser from the Microsoft Web site. If you intend to persist your files locally in XML format, be aware of a few disadvantages. First, the saving and loading of XML files is slower than the saving and loading of ADTG files. Second, ADO's XML files (and XML files in general) are significantly larger than their ADTG counterparts (XML files are typically twice as large as their ADTG counterparts). Third, ADO's XML format is specific to Microsoft, as most companies' XML implementations are. This means that the XML generated in ADO is not readable by Borland's TClientDataSet and vice versa. In fact, it's worse than that, because the XML generated in ADO 2.1 is incompatible with the XML in ADO 2.5. Fortunately this last problem can be overcome using Delphi 6's new TXMLTransform component, which can be used to translate between different XML structures.

If you intend to use these features solely for single-tier applications and not as part of the briefcase model, then you can save yourself a little effort by using a TADODataSet and setting its CommandType to cmdFile and its CommandText to the name of the file. This will save you the effort of having to call LoadFromFile manually. You will, however, still have to call SaveToFile. In a briefcase application, however, this approach is too limiting, as the dataset can be used in two different modes.

The Briefcase Model

Our new-found knowledge of batch updates, disconnected recordsets, and persistent recordsets allows us to take advantage of the "briefcase model." The idea behind the briefcase model is that your users want to be able to use your application while they are out on the road. They want to take the same application that they use on the desktops in the office and use it on their laptops while on their clients' sites. The problem with this scenario traditionally is that when your users are at their clients' sites, they are not connected to their database server, because their database server is running on their network back at their office. Consequently, there is no data on their laptop, and the data cannot be updated anyway.

This is where that new-found knowledge comes in. Assume that the application has already been written; the user has requested this new briefcase enhancement, and you have to retrofit it into your existing application. You need to add a new option for your users to allow them to "prepare" the briefcase application. This simply consists of executing `SaveToFile` for each and every table in the database. The result is a collection of ADTG or XML files that mirror the contents of the database. These files are then copied to the laptop where a copy of the application has previously been installed.

The application needs to be sensitive to whether it is running locally or connected to the network. You can decide this either by attempting to connect to the database and seeing whether it fails, by detecting the presence of a local "briefcase" file, or by creating some flag of your own design. If the application decides it is running in briefcase mode, then it needs to use `LoadFromFile` for each table instead of setting `Connected` to True for the TADOConnections and `Active` to True for the ADO datasets. Thereafter, the briefcase application needs to use `SaveToFile` instead of `UpdateBatch` whenever data is saved. Upon return to the office, there needs to be an update process where each table is loaded from its local file, the dataset is connected to the database, and the changes are applied using `UpdateBatch`. Voilà, the briefcase model.

Deploying MDAC

MDAC, and therefore ADO, can be almost freely distributed. There are some conditions on its distribution, but these are to protect Microsoft from unreasonable behavior and it is unlikely that regular application developers will fall afoul of them. To distribute MDAC, you distribute and execute `MDAC_TYP.EXE`. You may not distribute and install MDAC components individually. If you use InstallShield Express For Delphi, then you will have to run `MDAC_TYP.EXE` separately from your regular InstallShield Setup program, because InstallShield Express For Delphi cannot shell out to an external program—i.e., `MDAC_TYP.EXE`. If you use an installation program that can shell out to another program, you might want to be aware of some of the parameters you can pass to MDAC_TYP.EXE. The various parameters affect whether the user has to specifically accept the end user license, whether the file copy dialog is shown, whether there is an automatic reboot on completion (or the user decides to reboot or there is no reboot), and, finally, whether MDACSET.LOG is a summary log file or a full log file. The /Q parameter is a quiet, but not completely silent, setup. A truly silent setup can be achieved with /Q:A /C:"setup /QNT" parameters. See the Platform SDK for a complete list of setup parameters. Although I have not tested installing every version of MDAC on top of every other version of MDAC, my experience is that, despite the progress bar indicating the successful progress of the installation, an earlier version of MDAC does not overwrite a later version of MDAC.

In addition to installing MDAC, you will also need to install DCOM if the target is Windows 95.

One invaluable tool that you should add to your toolbox is Component Checker. This is available for free download from www.microsoft.com/data/download.htm. Component Checker is the most accurate method of determining which version of MDAC is installed on a machine. It scans for all ADO, OLE DB, and ODBC files and gets their version numbers. It can compare all of these version numbers against its own internal database of correct version numbers for each release of MDAC. At the end of its analysis, it reveals the version of MDAC that most closely matches the files installed on a machine. It is also useful as a kind of "REGEDIT For MDAC," as it reports on all of the MDAC registry information using a considerably more relevant user interface than REGEDIT. Finally, it is the only safe way of removing MDAC from a machine.

ADO.NET

ADO.NET is part of Microsoft's new dotNet (or ".NET") architecture—their redesign of application development tools to better suit the needs of Web development. At the time of writing, Visual Studio.NET was in beta, so this section has been included solely as a means to give you an idea of where ADO is heading.

ADO.NET is a revolution of ADO. It looks at the problems of Web development and addresses shortcomings of ADO's solution. The problem with ADO's solution is that it is based on COM. For one- and two-tier applications, COM imposes few problems, but in the world of Web development it is unacceptable as a transport mechanism. COM suffers from three main problems for use in Web development: it (mostly) runs only on Windows; the transmission of recordsets from one process requires COM marshalling; and COM calls cannot penetrate corporate firewalls. ADO.NET's solution to all of these problems is to use XML.

Continued on next page

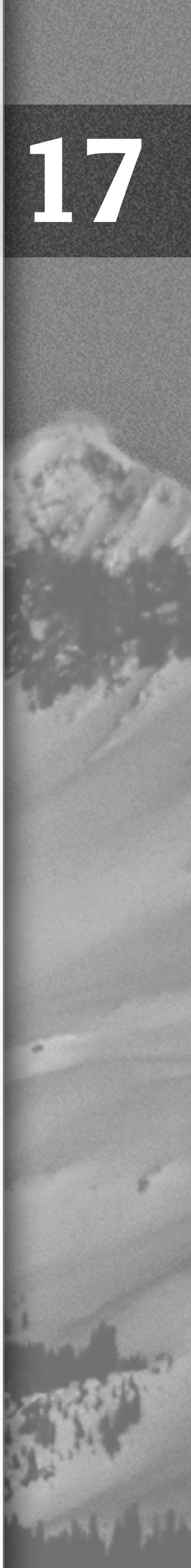
Some other redesign issues focus around breaking up the ADO recordset into separate classes. The resulting classes are adept at solving a single problem instead of multiple problems. For example, the ADO.NET class currently called `DataSetReader` is very similar to a read-only, forward-only server-side recordset and, as such, is best suited to reading a result set very quickly. A `DataTable` is most like a disconnected, client-side recordset. A `DataRelation` has similarities with the MSDataShape OLE DB Provider. So you can see that your knowledge of how ADO works is of great benefit in understanding the basic principles of ADO.NET.

If you wish to experiment with ADO.NET in Delphi before direct support is added to Delphi, then you will need to give Delphi access to the ADO.NET classes, which are called "managed" classes and are based on a new run-time environment called the Common Language Runtime (CLR). These classes are not COM classes and, as such, Delphi cannot normally access them. However, Visual Studio.NET includes a utility called `REGASM.EXE`, which takes any "assembly" (library of managed classes) and creates a COM type library interface to the managed classes. As Delphi can easily access COM classes, simply import the resulting type library into Delphi and use the classes as if they were COM classes.

What's Next?

This chapter described ActiveX Data Objects (ADO) and dbGo, the set of Delphi components for accessing the ADO interfaces. You've seen how to take advantage of Microsoft Data Access Components (MDAC) and various server engines, and I've described some of the benefits and hurdles you'll encounter in using ADO.

The next chapter will take you into the world of Delphi's DataSnap architecture, for developing custom client and server applications in a three-tier environment.



CHAPTER 17

Multitier Database Applications with DataSnap

- Logical three-tier architecture
- The technical foundation of DataSnap
- The connection protocols and the data packets
- Delphi's support components (client-side and server-side)
- The connections broker and other new Delphi 6 features

Large companies often have broader needs than applications using local database and SQL servers can meet. In the past few years, Borland Software Corporation has been addressing the needs of large corporations, and it even temporarily changed its own name to Inprise to underline this new enterprise focus. The name was changed back to Borland, but the focus on enterprise development remains.

Delphi is targeting many different technologies: three-tier architectures based on Windows NT and DCOM, CORBA architectures based on NT and Unix servers, TCP/IP and socket applications, and—most of all—SOAP- and XML-based Web services. This chapter focuses on database-oriented multitier architectures, while XML-oriented solutions will be discussed in Chapter 23, "XML and SOAP."

Even though I haven't yet discussed COM and sockets (covered in Chapters 19 to 21), in this chapter we'll build multitier architectures based on those technologies. As we'll use high-level Delphi support, not knowing the details of some of the foundations should not create any problem. I'll concentrate more on the programming aspects of these architectures than on installation and configuration (the latter aspects are subject to change across different operating systems and are too complex to cover thoroughly).

Before proceeding, I should emphasize two important elements. First, the tools to support this kind of development are available only in the Enterprise version of Delphi; and second, you'll have to pay a license fee to Borland in order to deploy the necessary server-side software for DataSnap. This second requirement makes this architecture cost-effective mainly for large systems (that is, servers connected to dozens or even hundreds of clients). The license fee is only required for deployment of the server application and is a flat fee for each server you deploy to (regardless of the number of clients that will connect). The license fee is not required for development or evaluation purposes.

NOTE You spend money on the DataSnap license, but you might save on the SQL server client licenses. When SQL server licenses were based on the number of connections, companies have saved tens of thousands of dollars in those licenses by connecting the hundreds of clients to a few instances of the DataSnap server, using few connections with the SQL server. Nowadays, the licenses for most SQL servers are based on the number of users who connect to the database, not the number of connections active at each time, so this kind of savings doesn't always apply.

One, Two, Three Levels

Initially, database PC applications were client-only solutions: the program and the database files were on the same computer. From there, adventuresome programmers moved the database files onto a network file server. The client computers still hosted the application software and the entire database engine, but the database files were now accessible to several users at the same time. You can still use this type of configuration with a Delphi application and Paradox files (or, of course, Paradox itself), but the approach was much more widespread just few years ago.

The next big transition was to client/server development, embraced by Delphi since its first version. In the client/server world, the client computer requests the data from a server computer, which hosts both the database files and a database engine to access them. This architecture downplays the role of the client, but it also reduces its requirements for processing power on the client machine. Depending on how the programmers implement client/server, the server can do most (if not all) of the data processing. In this way, a powerful server can provide data services to several less powerful clients.

Naturally, there are many other reasons for using centralized database servers, such as the concern for data security and integrity, simpler backup strategies, central management of data constraints, and so on. The database server is often called a SQL server, because SQL is the language most commonly used for making queries into the data, but it may also be called a DBMS (database management system), reflecting the fact that the server provides tools for managing the data, such as support for backup and replication.

Of course, some applications you build may not need the benefits of a full DBMS, so a simple client-only solution might be sufficient. On the other hand, you might need some of the robustness of a DBMS system, but on a single, isolated computer. In this case, you can use a local version of a SQL server, such as InterBase. Traditional client/server development is done with a two-tier architecture. However, if the DBMS is primarily performing data storage instead of data- and number-crunching, the client might contain both user interface code (formatting the output and input with customized reports, data-entry forms, query screens, and so on) and code related to managing the data (also known as *business rules*). In this case, it's generally a good idea to try to separate these two sections of the program and build a logical three-tier architecture. The term *logical* here means that there are still just two computers (that is, two physical tiers), but we've now partitioned the application into three distinct elements.

Delphi 2 introduced support for a logical three-tier architecture with data modules. As you'll recall, a *data module* is a nonvisual container for the data access components of an application, but it often includes several handlers for database-related events. You can share a single data module among several different forms and provide different user interfaces for the same data; there might be one or more data-input forms, reports, master/detail forms, and various charting or dynamic output forms.

The logical three-tier approach solves many problems, but it also has a few drawbacks. First, you must replicate the data-management portion of the program on different client computers, which might hamper performance, but a bigger issue is the complexity this adds to code maintenance. Second, when multiple clients modify the same data, there's no simple way to handle the resulting update conflicts. Finally, for logical three-tier Delphi applications, you must install and configure the database engine (if any) and SQL server client library on every client computer.

The next logical step up from client/server is to move the data-module portion of the application to a separate server computer and design all the client programs to interact with it. This is exactly the purpose of remote data modules, which were introduced in Delphi 3. Remote data modules run on a server computer—generally called the application server. The application server in turn communicates with the DBMS (which can run on the application server or on another dedicated computer). Therefore, the client machines don't connect to the SQL server directly, but indirectly via the application server.

At this point there is a fundamental question: Do we still need to install the database access software? The traditional Delphi client/server architecture (even with three logical tiers) requires you to install the database access on each client, something quite troublesome when you must configure and maintain hundreds of machines. In the physical three-tier architecture, you need to install and configure the database access only on the application server, not on the client computers. Since the client programs have only user interface code and are extremely simple to install, they now fall into the category of so-called *thin clients*. To use marketing-speak, we might even call this a *zero-configuration thin-client architecture*. But let us focus on technical issues instead of marketing terminology.

The Technical Foundation of DataSnap

When Borland introduced this physical multitier architecture in Delphi, it was called MIDAS (Middle-tier Distributed Application Services). For example, Delphi 5 included the third version of this technology, MIDAS 3. Now Delphi 6 renames this technology as *DataSnap* and extends its capabilities.

DataSnap requires the installation of specific libraries on the server (actually the middle-tier computer), which provides your client computers with the data extracted from the SQL server database or other data sources. DataSnap does not require a SQL server for data storage. DataSnap can serve up data from a wide variety of sources, including SQL, CORBA, other DataSnap servers, or just data computed on the fly.

As you would expect, the client side of DataSnap is extremely thin and easy to deploy. The only file you need is `Midas.dll`, a small (260 KB) DLL that implements the `ClientDataSet` and `RemoteServer` components and provides the connection to the application server. As we've seen in Chapter 14, "Client/Server Programming," this DLL is basically a small, stand-alone database engine. It caches data from a remote data module and enforces the rules requested by the Constraint Broker.

The application server uses the same DLL to handle the datasets (called *deltas*) returned from the clients when they post updated or new records. However, the server also requires several other libraries, all of which are installed by DataSnap.

The *IAppServer* Interface

Starting with Delphi 5, the two sides of a DataSnap application communicate using the `IAppServer` interface.

NOTE In Delphi 5 (and 6), the `IAppServer` interface supersedes Delphi 4's `IProvider` interface. The main reason for this change was support for stateless objects. With `IProvider`, the server stored status information about the client program—for example, which records had already been passed to the client. This made it difficult to adapt the server-side objects to stateless connection layers, like CORBA message queues and MTS, and also to move toward HTTP and Web-based support. Other reasons for moving to this new architecture were to make the system more dynamic (providers are now exported by setting a property, not by changing the type library) and to reduce the number of calls, or *round-trips*, which can affect performance. DataSnap makes fewer calls but delivers more data each time.

The `IAppServer` interface has the following methods:

```
AS_ApplyUpdates
AS_GetRecords
AS_DataRequest
AS_GetProviderNames
AS_GetParams
AS_RowRequest
AS_Execute
```

You'll seldom need to call them directly, anyway, because there are Delphi components to be used on the client and server sides of the application that embed these calls, making them easier (and at times even hiding them completely). In practice, the server will make available to the client objects implementing this interface, possibly along with other custom interfaces.

NOTE A DataSnap server exposes an interface using a COM type library, a technology I'll discuss in Chapter 20, "Automation, ActiveX, and Other COM Technologies."

The Connection Protocol

DataSnap defines only the higher-level architecture and can use different technologies for moving the data from the middle tier to the client side. DataSnap supports most of the leading standards, including the following:

Distributed COM (DCOM) and Stateless COM (MTS or COM+) DCOM is directly available in Windows NT/2000 and 98/Me, and it requires no additional run-time applications on the server. You still have to install it on Windows 95 machines. DCOM is basically an extension of COM technology (discussed in Chapter 19, "COM Programming," and Chapter 20) that allows a client application to use server objects that exist and execute on a separate computer. The DCOM infrastructure allows you to use stateless COM objects, available in the COM+ and in the older MTS (Microsoft Transaction Server) architectures. Both COM+ and MTS provide features such as security, component management, and database transactions, and are available in Windows NT/2000 and in Windows 98/Me.

Due to the complexity of DCOM configuration and of its problems in passing through firewalls, even Microsoft is abandoning DCOM in favor of SOAP-based solutions.

TCP/IP sockets These are available on most systems. Using TCP/IP you might distribute clients over the Web, where DCOM cannot be taken for granted, and have many fewer configuration headaches. To use sockets, the middle-tier computer must run the `ScktSrvr.exe` application provided by Borland, a single program that can run either as an application or as a service. This program receives the client requests and forwards them to the remote data module (executing on the same server) using COM. Sockets provide no protection against failure on the client side, as the server is not informed and might not release resources when a client unexpectedly shuts down.

HTTP and SOAP The use of HTTP as a transport protocol over the Internet simplifies connections through firewalls or proxy servers (which generally don't like custom TCP/IP sockets). You need a specific Web server application, `httpsrvr.dll`, which accepts client requests and creates the proper remote data modules using COM. These Web connections can use SSL security but must register themselves by adding a call to `EnableWebTransport` in the `UpdateRegistry` method. Finally, Web connections based on HTTP transport can use DataSnap object-pooling support.

NOTE The DataSnap HTTP transport can use XML as the data packet format, enabling any platform or tool that can read XML to participate in a DataSnap architecture. This is an extension of the original DataSnap data packet format, which is also platform-independent. The use of XML over HTTP is also the foundation of SOAP. There's more on XML in Chapter 23.

CORBA Common Object Request Broker Architecture is an official standard for object management available on most operating systems. Compared to DCOM, the advantage is that your client and server applications can be also written with Java and other products. The Borland implementation of CORBA, VisiBroker, is available with Delphi Enterprise. CORBA provides many benefits, including location transparency, load balancing, and failover from the ORB run-time software. (An in-depth discussion of CORBA is certainly beyond the scope of this book, and in practice only a limited number of Delphi programmers use CORBA.)

Internet Express As an extension to this architecture, you can transform the data packets into XML and deliver them to a Web browser. In this case, you basically have one extra tier: the Web server gets the data from the middle tier and delivers it to the client. I'll discuss this new architecture, called Internet Express, in Chapter 23. The DLL can also be folded into the executable file by using the MidasLib unit.

Providing Data Packets

The entire Delphi multitier data-access architecture centers around the idea of *data packets*. In this context, a data packet is a block of data that moves from the application server to the client or from the client back to the server. Technically, a data packet is a sort of subset of a dataset. It describes the data it contains (usually a few records of data), and it lists the names and types of the data fields. Even more important, a data packet includes the constraints—that is, the rules to be applied to the dataset. You'll typically set these constraints in the application server, and the server sends them to the client applications along with the data.

All communication between the client and the server occurs by exchanging data packets. The provider component on the server manages the transmission of several data packets within a big dataset, with the goal of responding faster to the user. As the client receives a data packet, in a ClientDataSet component, the user can edit the records it contains. As mentioned earlier, during this process the client also receives and checks the constraints, which are applied during the editing operations.

When the client has updated the records and sends a data packet back, that packet is known as a *delta*. The delta packet tracks the difference between the original records and the updated ones, recording all the changes the client requested from the server. When the client asks to apply the updates to the server, it sends the delta to the server, and the server tries to apply each of the changes. I say *tries* because if a server is connected to several clients, the data might have changed already, and the update request might fail.

Since the delta packet includes the original data, the server can quickly determine if another client has already changed it. If so, the server fires an `OnReconcileError` event, which is one of the vital elements for thin-client applications. In other words, the three-tier architecture uses an update mechanism similar to the one Delphi uses for cached updates. As we have seen in Chapter 14, "Client/Server Programming Techniques," the `ClientDataSet` manages data in a memory cache, and it typically reads only a subset of the records available on the server side, loading more elements only as they're needed. When the client updates records or inserts new ones, it stores these pending changes in another local cache on the client, the *delta cache*.

The client can also save the data packets to disk and work off-line, thanks to the MyBase support discussed in Chapter 13, "Delphi's Database Architecture." Even error information and other data moves using the data packet protocol, so it is truly one of the foundation elements of this architecture.

NOTE It's important to remember that data packets are protocol-independent. A data packet is merely a sequence of bytes, so anywhere you can move a series of bytes, you can move a data packet. This was done to make the architecture suitable for multiple transport protocols (including DCOM, CORBA, HTTP, and TCP/IP) and for multiple platforms.

Delphi Support Components (Client-Side)

Now that we've examined the general foundations of Delphi's three-tier architecture, we can focus on the components that support it. For developing client applications, Delphi provides the ClientDataSet component, which provides all the standard dataset capabilities and embeds the client side of the `IAppServer` interface. In this case, the data is delivered through the remote connection.

The connection to the server application is made via another component you'll also need in the client application. You should use one of the four specific connection components (available in the DataSnap page):

- The DCOMConnection component can be used on the client side to connect to a DCOM and MTS server, located either on the current computer or in another one indicated by the `ComputerName` property. The connection is with a registered object having a given `ServerGUID` or `ServerName`.
- The CorbaConnection component can be used to hook with a CORBA server. You indicate the `HostName` (the name or IP address) to indicate the server computer, the `RepositoryID` to request a specific data module on the server, and optionally the `ObjectName` property if the data module exports multiple objects.
- The SocketConnection component can be used to connect to the server via a TCP/IP socket. You should indicate the IP address or the host name, and the GUID of the server object (in the `InterceptGUID` property). In Delphi 5, this connection component

has an extra property, SupportCallbacks, which you can disable if you are not using callbacks and want to deploy your program on Windows 95 computers that don't have Winsock 2 installed.

NOTE In the WebServices page, you can also find the SoapConnection component, which requires a specific type of server and will be discussed in Chapter 23.

- The WebConnection component is used to handle an HTTP connection that can easily get through a firewall. You should indicate the URL where your copy of httpsrvr.dll is located and the name or GUID of the remote object on the server.

Delphi 6 adds new client-side components to the DataSnap architecture, mainly for managing connections:

- The ConnectionBroker component can be used as an alias of an actual connection component, something useful when you have a single application with multiple client datasets. In fact, to change the physical connection of each of the datasets, you only need to change the Connection property of the ConnectionBroker. You can also use the events of this virtual connection component in place of those of the actual connections, so you don't have to change any code if you change the data transport technology. For the same reason, you can refer to the AppServer object of the ConnectionBroker instead of the corresponding property of a physical connection.
- The SharedConnection component can be used to connect to a secondary (or child) data module of a remote application, piggy-backing on an existing physical connection to the main data module. In other words, an application can connect to multiple data modules of the server with a single, shared connection.
- The LocalConnection component can be used to target a local dataset provider as the source of the data packet. The same effect can be obtained by hooking the ClientDataSet directory to the provider. However, using the LocalConnection, you can write a local application with the same code as a complete multitier application, using the IAppServer interface of the "fake" connection. This will make the program easier to scale up, compared to a program with a direct connection.

A few other components of the DataSnap page relate to the transformation of the DataSnap data packet into custom XML formats. These components (XMLTransform, XMLTransform-Provider, and XMLTransformClient) will be discussed in Chapter 23.

Delphi Support Components (Server-Side)

On the server side (or actually the middle tier), you'll need to create an application or a library that embeds a remote data module, a special version of the `TDataModule` class. There are actually specialized remote data modules for transactional COM and CORBA support. In the Multitier page of the New Items dialog box (obtained with the File ➢ New ➢ Others menu) are specific wizards to create remote data modules of each of these types.

The only specific component you need on the server side is the DataSetProvider. You need one of these components for every table or query you want to make available to the client applications, which will then use a separate ClientDataSet component for every exported dataset. The DataSetProvider was already introduced in Chapter 14.

NOTE The DataSetProvider component of Delphi 5 and 6 supersedes the stand-alone Provider component of Delphi 4 and the internal Provider object, which was embedded in the `TBDEDataSet` subclasses.

Building a Sample Application

Now we're ready to build a sample program. This will allow us to observe some of the components I've just described in action, and it will also allow us to focus on some other problems, shedding light on other pieces of the Delphi multitier puzzle. I'll build the client and application server portions of a three-tier application in two steps. The first step will simply test the technology using a bare minimum of elements. These programs will be very simple.

From that point, we'll add more power to the client and the application server. In each of the examples, we'll display data from local Paradox tables, and we'll set up everything to allow you to test the programs on a stand-alone computer. I won't cover the steps you have to follow to install the examples on multiple computers with various technologies—that would be the subject of at least one other book.

The First Application Server

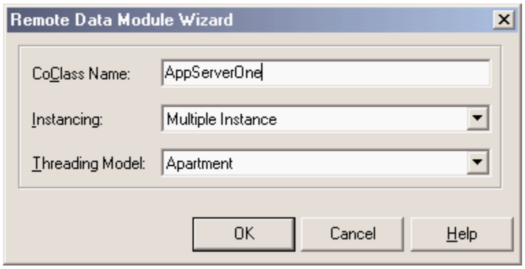
The server side of our basic example is very easy to build. Simply create a new application and add a remote data module to it using the corresponding icon in the Multitier page of the Object Repository. The simple Remote Data Module Wizard (see Figure 17.1) will ask you for a class name and the instancing style. As you enter a class name, such as AppServerOne, and click the OK button, Delphi will add a data module to the program. This data module will have the usual properties and events, but its class will have the following Pascal declaration:

```
type
  TAppServerOne = class(TRemoteDataModule, IAppServerOne)
  private
```

```
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
  public
    { Public declarations }
  end;
```

FIGURE 17.1:

The simple Remote Data Module Wizard


Remote Data Module Wizard

CoClass Name: AppServerOne

Instancing: Multiple Instance

Threading Model: Apartment

In addition to inheriting from the TRemoteDataModule base class, this class implements the custom IAppServerOne interface, which derives from the standard DataSnap interface (IAppServer). The class also overrides the UpdateRegistry method to add the support for enabling the socket and Web transports, as you can see in the code generated by the wizard. At the end of the unit, you'll find the class factory declaration, which will become clear after reading Chapter 19:

```
initialization
  TComponentFactory.Create(ComServer, TAppServerOne,
    Class_AppServerOne, ciMultiInstance, tmApartment);
end.
```

Now you can add a dataset component to the data module (I've used the dbExpress SQLDataSet), connect it to a database and a table or query, activate it, and finally add a DataSetProvider and hook it to the dataset component. You'll obtain a DFM file like this:

```
object AppServerOne: TAppServerOne
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
    Constraints = True
    end
  end
```

What about the main form of this program? Well, it's almost useless, so we can simply add a label to it indicating that it's the form of the server application. When you've built the server, you should compile it and run it once. This operation will automatically register it as an Automation server on your system, making it available to client applications. Of course, you should register the server on the computer where you want it to run, either the client or the middle tier.

The First Thin Client

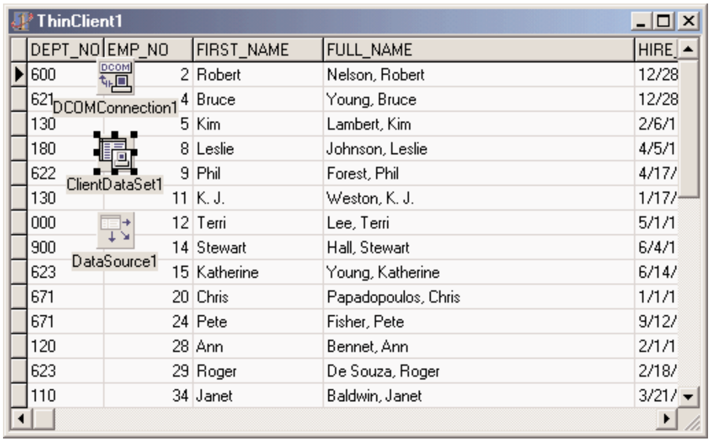
Now that we have a working server, we can build a client that will connect to it. We'll again start with a standard Delphi application and add a DCOMConnection component to it (or the proper component for the specific type of connection you want to test). This component defines a `ComputerName` property that you'll use to specify the computer that hosts the application server. If you want to test the client and application server from the same computer, you can leave this blank.

Once you've selected an application server computer, you can simply display the `ServerName` property's combo-box list to view the available DataSnap servers. This combo box shows the servers' registered names, by default the name of the executable file of the server followed by the name of the remote data module class, as in `AppServ1.AppServerOne`. Alternatively, you can type the GUID of the server object in the `ServerGUID` property. Delphi will automatically fill this property as you set the `ServerName` property, determining the GUID by looking it up in the Registry.

At this point, if you set the DCOMConnection component's `Connected` property to True, the server form will appear, indicating that the client has activated the server. You don't usually need to perform this operation, because the ClientDataSet component typically activates the RemoteServer component for you. I've suggested this simply to emphasize what's happening behind the scenes.

TIP You should generally leave the DCOMConnection component's `Connected` property set to False at design time, to be able to open the project in Delphi even on a computer where the DataSnap server is not already registered.

As you might expect, the next step is to add a ClientDataSet component to the form. You must connect the ClientDataSet to the `DCOMConnection1` component via the `RemoteServer` property, and thereby to one of the providers it exports. You can see the list of available providers in the `ProviderName` property, via the usual combo box. In this example, you'll be able to select only `DataSetProvider1`, as this is the only provider available in the server we've just built. This operation connects the dataset in the client's memory with the dbExpress dataset on the server. If you activate the client dataset and add a few data-aware controls (or a DBGrid), you'll immediately see the server data appear in them, as illustrated in Figure 17.2.



DEPT_NO	EMP_NO	FIRST_NAME	FULL_NAME	HIRE_
600	2	Robert	Nelson, Robert	12/28
621	4	Bruce	Young, Bruce	12/28
130	5	Kim	Lambert, Kim	2/6/1
180	8	Leslie	Johnson, Leslie	4/5/1
622	9	Phil	Forest, Phil	4/17/
130	11	K. J.	Weston, K. J.	1/17/
000	12	Terri	Lee, Terri	5/1/1
900	14	Stewart	Hall, Stewart	6/4/1
623	15	Katherine	Young, Katherine	6/14/
671	20	Chris	Papadopoulos, Chris	1/1/1
671	24	Pete	Fisher, Pete	9/12/
120	28	Ann	Bennet, Ann	2/1/1
623	29	Roger	De Souza, Roger	2/18/
110	34	Janet	Baldwin, Janet	3/21/

FIGURE 17.2: When you activate a ClientDataSet component connected to a remote data module at design time, the data from the server becomes visible as usual.

Here is the DFM file for our minimal client application, ThinCli1:

```
object Form1: TForm1
  Caption = 'ThinClient1'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object DCOMConnection1: TDCOMConnection
    ServerGUID = '{09E11D63-4A55-11D3-B9F1-00000100A27B}'
    ServerName = 'AppServ1.AppServerOne'
  end
  object ClientDataSet1: TClientDataSet
    Aggregates = <>
    Params = <>
    ProviderName = 'DataSetProvider1'
    RemoteServer = DCOMConnection1
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
end
```

Obviously, the programs of our first three-tier application are quite simple, but they demonstrate how to create a dataset viewer that splits the work between two different executable files. At this point, our client is only a viewer. If you edit the data on the client, it won't be updated on the server. To accomplish this, you'll need to add some more code to the client. However, before we do that, let's add some features to the server.

Adding Constraints to the Server

When you write a traditional data module in Delphi, you can easily add some of the application logic, or business rules, by handling the dataset events, and by setting field object properties and handling their events. You should avoid doing this work on the client application; instead, write your business rules on the middle tier.

In the DataSnap architecture, you can send some constraints from the server to the client and let the client program impose those constraints during the user input. You can also send field properties (such as minimum and maximum values and the display and edit masks) to the client and (using some of the data access technologies) process updates through the dataset used to access the data (or a companion UpdateSql object).

Field and Table Constraints

When the provider interface creates data packets to send to the client, it includes the field definitions, the table and field constraints, and one or more records (as requested by the ClientDataSet component). This implies that you can customize the middle tier and build distributed application logic by using SQL-based constraints.

The constraints you create using SQL expressions can be assigned to an entire dataset or to specific fields. The provider sends the constraints to the client along with the data, and the client applies them before sending updates back to the server. This reduces network traffic, compared to having the client send updates back to the application server and eventually up to the SQL server, only to find that the data is invalid. Another advantage of coding the constraints on the server side is that if the business rules change, you need to update the single server application and not the many clients on multiple computers.

But how do you write constraints? There are several properties you can use:

- BDE datasets have a `Constraints` property, which is a collection of `TCheckConstraint` objects. Every object has a few properties, including the expression and the error message.
- Each field object defines the `CustomConstraint` and `ConstraintErroMessage` properties. There is also an `ImportedConstraint` property for constraints imported from the SQL server.
- Each field object has also a `DefaultExpression` property, which can be used locally or passed to the ClientDataSet. This is not an actual constraint, only a suggestion to the end user.

Our next example, AppServ2, adds a few constraints to a remote data module connected to the sample EMPLOYEE InterBase database. After connecting the table to the database and creating the field objects for it, you can set the following special properties:

```
object SQLDataSet1: TSQLDataSet
  ...
  object SQLDataSet1EMP_NO: TSmallintField
    CustomConstraint = 'x > 0 and x < 10000'
    ConstraintErrorMessage =
      'Employee number must be a positive integer below 10000'
    FieldName = 'EMP_NO'
  end
  object SQLDataSet1FIRST_NAME: TStringField
    CustomConstraint = 'x <> '#39#39
    ConstraintErrorMessage = 'The first name is required'
    FieldName = 'FIRST_NAME'
    Size = 15
  end
  object SQLDataSet1LAST_NAME: TStringField
    CustomConstraint = 'not x is null'
    ConstraintErrorMessage = 'The last name is required'
    FieldName = 'LAST_NAME'
  end
end
```

NOTE The expression `'x <> '#39#39` is the DFM transposition of the string `x <> ''`, indicating that we don't want to have an empty string. The final constraint, `not x is null`, instead allows empty strings but not null values.

Including Field Properties

You can control whether the properties of the field objects on the middle tier are sent to the ClientDataSet (and copied into the corresponding field objects of the client side), by using the `poIncFieldProps` value of the `Options` property of the DataSetProvider. This flag controls the download of the field properties `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, and `DisplayValues`, if they are available in the field. Here is an example of another field of the AppServ2 example with some custom properties:

```
object SQLDataSet1SALARY: TBCDField
  DefaultExpression = '10000'
  FieldName = 'SALARY'
  DisplayFormat = '#,###'
  EditFormat = '####'
  Precision = 15
  Size = 2
end
```

With this setting, you can simply write your middle tier the way you usually set the fields of a standard client/server application. This approach also makes it faster to move existing applications from a client/server to a multitier architecture. The main drawback of sending fields to the client is that transmitting all the extra information takes time. Turning off `poIncFieldProps` can dramatically improve network performance of datasets with many columns.

A server can generally filter the fields returned to the client; it does this by declaring persistent field objects with the Fields editor and omitting some of the fields. Because a field you're filtering out might be required to identify the record for future updates (if the field is part of the primary key), you can also use the field's `ProviderFlags` property on the server to send the field value to the client but make it unavailable to the ClientDataSet component (this provides some extra security, compared to sending the field to the client and hiding it there).

Field and Table Events

You can write middle-tier dataset and field event handlers as usual and let the dataset process the updates received by the client in the traditional way. This means that updates are considered to be operations on the dataset, exactly as when a user is directly editing, inserting, or deleting fields locally.

This is accomplished by setting the `ResolveToDataSet` property of the `TDatasetProvider` component, again connecting either the dataset used for input or a second one used for the updates. This approach is possible with datasets supporting editing operations. These includes BDE, ADO, and InterBase Express datasets, but not those of the new dbExpress architecture.

With this technique, the updates are performed by the dataset, which implies a lot of control (the standard events are being triggered) but generally slower performance. Flexibility is much greater, as you can use standard coding practices. Also, porting existing local or client/server database applications, which use dataset and field events, is much more straightforward with this model. However, keep in mind that the user of the client program will receive your error messages only when the local cache (the delta) is sent back to the middle tier. Saying to the user that some data prepared half an hour ago is not valid might be a little awkward. If you follow this approach, you'll probably need to apply the updates in the cache at every `AfterPost` event on the client side.

Finally, if you decide to let the dataset and not the provider do the updates, Delphi helps you a lot in handling possible exceptions. Any exceptions raised by the middle-tier update events (for example, `OnBeforePost`) are automatically transformed by Delphi into update errors, which activate the `OnReconcileError` event on the client side (more on this event later in this chapter). No exception is shown on the middle tier, but the error travels back to the client.

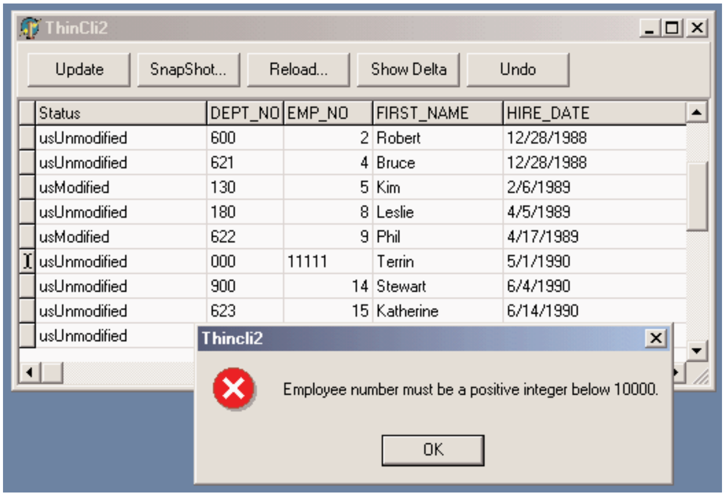
Adding Features to the Client

After adding some constraints and field properties to the server, we can now return our attention to the client application. The first version was very simple, but now there are several features we can add to it to make it work well. In the ThinCli2 example, I've embedded support for checking the record status and accessing the delta information (the updates to be sent back to the server), using some of the ClientDataSet techniques already discussed in Chapter 13. The program also handles reconcile errors and supports the briefcase model.

Keep in mind that while you're using this client to edit the data locally, you'll be reminded of any failure to match the business rules of the application, set up on the server side using constraints. The server will also provide us with a default value for the Salary field of a new record and pass along the value of its `DisplayFormat` property. In Figure 17.3 you can see one of the error messages this client application can display, which it receives from the server. This message is displayed while editing the data locally, not when you send it back to the server.

FIGURE 17.3:

The error message displayed by the ThinCli2 example when the employee ID is too large



Status	DEPT_NO	EMP_NO	FIRST_NAME	HIRE_DATE
usUnmodified	600	2	Robert	12/28/1988
usUnmodified	621	4	Bruce	12/28/1988
usModified	130	5	Kim	2/6/1989
usUnmodified	180	8	Leslie	4/5/1989
usModified	622	9	Phil	4/17/1989
usUnmodified	000	11111	Terrin	5/1/1990
usUnmodified	900	14	Stewart	6/4/1990
usUnmodified	623	15	Katherine	6/14/1990
usUnmodified				

Employee number must be a positive integer below 10000.

The Update Sequence

This client program also includes a button to Apply the updates to the server and a standard reconcile dialog. Here is a summary of the complete sequence of operations related to an update request and the possible error events:

1. The client program calls the `ApplyUpdates` method of a ClientDataSet.
2. The delta is sent to the provider on the middle tier. The provider fires the `OnUpdateData` event, where you have a chance to look at the requested changes before they reach the

database server. At this point you can modify the delta, which is passed in a format compatible with the data of a ClientDataSet.

3. The provider (technically, a part of the provider called the "resolver") applies each row of the delta to the database server. Before applying each update, the provider receives a `BeforeUpdateRecord` event. If you've set the `ResolveToDataSet` flag, this update will eventually fire local events of the dataset in the middle tier.

4. In case of a server error, the provider fires the `OnUpdateError` event (on the middle tier) and the program has a chance of fixing the error at that level.

5. If the middle-tier program doesn't fix the error, the corresponding update request remains in the delta. The error is returned to the client side at this point or after a given number of errors have been collected, depending on the value of the `MaxErrors` parameter of the `ApplyUpdates` call.

6. Finally, the delta packet with the remaining updates is sent back to the client, firing the `OnReconcileError` event of the ClientDataSet for each remaining update. In this event handler, the client program can try to fix the problem (possibly prompting the user for help), modifying the update in the delta, and later reissuing it.

Refreshing Data

You can obtain an updated version of the data, which other users might have modified, by calling the `Refresh` method of the ClientDataSet. However, this operation can be done only if there are no pending update operations in the cache, as calling `Refresh` raises an exception when the change log is not empty:

```
if cds.ChangeCount = 0 then
  cds.Refresh;
```

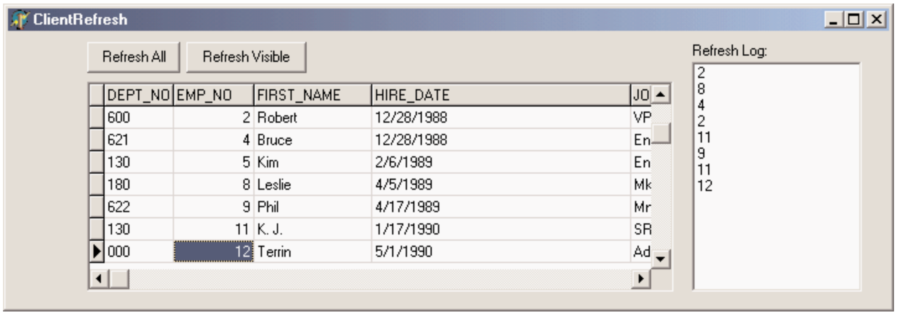
If only some records have been changed, you can refresh the others by calling `RefreshRecords`. This method refreshes only the current record, but it should be used only if the user hasn't modified the current record. In this case, in fact, `RefreshRecords` leaves the unapplied changes in the change log. As an example, you can refresh a record every time it becomes the active one, unless it has been modified and the changes have not yet been posted to the server:

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);
begin
  if cds.UpdateStatus = usUnModified then
    cds.RefreshRecord;
end;
```

When the data is subject to frequent changes by many users and each user should see changes right away, you should generally apply any change immediately in the `AfterPost` and `AfterDelete` methods, and call `RefreshRecords` for the active record (as shown above) or each of the records visible inside a grid. This code is actually part of the ClientRefresh example, connected to the AppServ2 server. For debugging purposes, the program also logs the EMP_NO field for each record it refreshes, as you can see in Figure 17.4.

FIGURE 17.4:

The form of the ClientRefresh example, which automatically refreshes the active record and allows more extensive updates by pressing the buttons



DEPT_NO	EMP_NO	FIRST_NAME	HIRE_DATE	JO
600	2	Robert	12/28/1988	VP
621	4	Bruce	12/28/1988	En
130	5	Kim	2/6/1989	En
180	8	Leslie	4/5/1989	Mk
622	9	Phil	4/17/1989	Mr
130	11	K. J.	1/17/1990	SR
000	12	Terrin	5/1/1990	Ad

```
2
8
4
2
11
9
11
12
```

I've done this by adding a button to the ClientRefresh example. The handler of this button moves from the current record to the first visible record of the grid and then to the last visible record. This is accomplished by noting that there are `RowCount - 1` rows visible, assuming that the first row is the fixed one hosting the field names. The program doesn't call `RefreshRecord` every time, as each movement will trigger an `AfterScroll` event with the code shown above. This is the code to refresh the visible rows, which might even be triggered by a timer:

```
var
  i: Integer;
  bm: TBookmarkStr;
begin
  // refresh visible rows
  cds.DisableControls;
  // start with the current row
  i := TMyGrid(DbGrid1).Row;
  bm := cds.Bookmark;
  try
    // get back to the first visible record
    while i > 1 do
    begin
      cds.Prior;
      Dec (i);
    end;
    // return to the current record
    i := TMyGrid(DbGrid1).Row;
    cds.Bookmark := bm;
    // go ahead until the grid is complete
    while i < TMyGrid(DbGrid1).RowCount do
    begin
      cds.Next;
      Inc (i);
    end;
  finally
    // set back everything and refresh
    cds.Bookmark := bm;
    cds.EnableControls;
  end;
```

This approach generates a huge amount of network traffic, so you might want to trigger updates only when there are actual changes. This can be implemented by adding a callback technology to the server, so that it can inform all connected clients that a given record has changed. The client can determine whether it is interested in the change and eventually trigger the update request.

Advanced DataSnap Features

There are many more features in DataSnap than I've covered up to now. Here is a quick tour of some of the more advanced features of the architecture, partially demonstrated by the AppSPlus and ThinPlus examples. Unfortunately, demonstrating every single idea would turn this chapter into an entire book (and not every Delphi programmer is interested in and can afford DataSnap), so I'll limit myself to an overview.

Besides the features discussed in the following sections, the AppSPlus and ThinPlus examples demonstrate the use of a socket connection, limited logging of events and updates on the server side, and direct fetching of a record on the client side. The last feature is accomplished with this call:

```
procedure TClientForm.ButtonFetchClick(Sender: TObject);
begin
  ButtonFetch.Caption := IntToStr (cds.GetNextPacket);
end;
```

This allows you to get more records than are actually required by the client user interface (the DBGrid). In other words, you can fetch records directly, without waiting for the user to scroll down in the grid. I suggest you study the details of these complex examples after reading the rest of this section.

ThinPlus example: This program requires Delphi's socket server (provided in Delphi's bin folder) to run. This is apparently not clear in the text. Without this program you'll see a socket error.

Parametric Queries

If you want to use parameters in a query or stored procedure, then instead of building a custom solution (with a custom method call to the server), you can let Delphi help you. First define the query on the middle tier with a parameter, such as:

```
select * from customer where Country = :Country
```

Use the `Params` property to set the type and default value of the parameter. On the client side, you can use the Fetch Params command of the ClientDataSet's shortcut menu, after connecting it to the proper provider. At run time, you can call the equivalent `FetchParams` method of the ClientDataSet component.

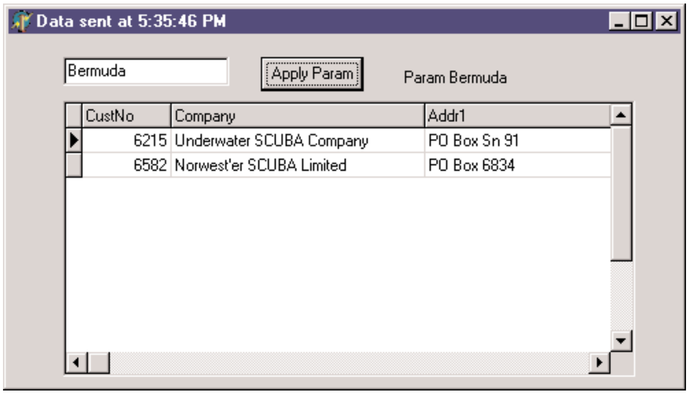
Now you can provide a local default value to the parameter by acting on the `Params` property. This will be sent to the middle tier when you fetch the data. The ThinPlus example refreshes the parameter with the following code:

```
procedure TFormQuery.btnParamClick(Sender: TObject);
begin
  cdsQuery.Close;
  cdsQuery.Params[0].AsString := EditParam.Text;
  cdsQuery.Open;
end;
```

You can see the secondary form of this example, which shows the result of the parametric query in a grid, in Figure 17.5. In the figure you can also see some custom data sent by the server, as explained in the section "Customizing the Data Packets."

FIGURE 17.5:

The secondary form of the ThinPlus example, showing the data of a parametric query



Data sent at 5:35:46 PM

Bermuda

Param Bermuda

CustNo	Company	Addr1
6215	Underwater SCUBA Company	PO Box Sn 91
6582	Norwest'er SCUBA Limited	PO Box 6834

Custom Method Calls

Since the server has a normal COM interface, we can add more methods or properties to it and call them from the client. Simply open the type library editor of the server and use it as with any other COM server. In the AppSPlus example, I've added a custom `Login` method with the following implementation:

```
procedure TAppServerPlus.Login(const Name, Password: WideString);
begin
  // TODO: add actual login code...
  if Password <> Name then
    raise Exception.Create ('Wrong name/password combination received')
  else
    Query.Active := True;
  ServerForm.Add ('Login:' + Name + '/' + Password);
end;
```

The program makes a simple test, instead of checking the name/password combination against a list of authorizations as a real application should do. Also, disabling the `Query` doesn't really work, as it can be activated by the provider. Disabling the DataSetProvider is actually a more robust approach. The client has a simple way to access the server, the `AppServer` property of the remote connection component. Here is a sample call from the ThinPlus example, which takes place in the `AfterConnect` event of the connection component:

```
procedure TClientForm.ConnectionAfterConnect(Sender: TObject);
begin
  Connection.AppServer.Login (Edit2.Text, Edit3.Text);
end;
```

Note that you can call extra methods of the COM interface through DCOM and also using a socket-based or HTTP connection. Because the program uses the `safecall` calling convention, the exception raised on the server is automatically forwarded and displayed on the client side. This way, when a user selects the Connect check box, the event handler used to enable the client datasets is interrupted, and a user with the wrong password won't be able to see the data.

NOTE Besides direct method calls from the client to the server, you can also implement callbacks from the server to the client. This can be used, for example, to notify every client of specific events. COM events are one way to do this. As an alternative, you can add a new interface, implemented by the client, which passes the implementation object to the server. This way, the server can call the method on the client computer. Callbacks are not possible with HTTP connections, though.

Master/Detail Relations

If your middle-tier application exports multiple datasets, you can retrieve them using multiple ClientDataSet components on the client side and connect them locally to form a master/detail structure. This will create quite a few problems for the detail dataset unless you retrieve all of the records locally.

This solution also makes it quite complex to apply the updates; you cannot usually cancel a master record until all related detail records have been removed, and you cannot add detail records until the new master record is properly in place. (Actually, different servers handle this differently, but in most cases where a foreign key is used, this is the standard behavior.) What you can do to solve this problem is to write complex code on the client side to update the records of the two tables according to the specific rules.

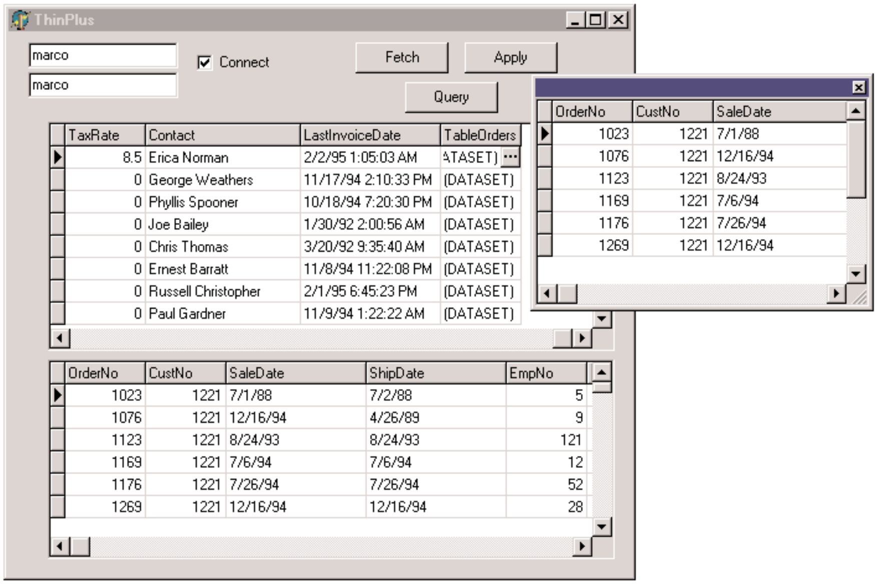
A completely different approach is to retrieve a single dataset that already includes the detail as a dataset field, a field of type TDatasetField. To accomplish this, you need to set up the master/detail relation on the server application:

```
object TableCustomer: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object TableOrders: TTable
  DatabaseName = 'DBDEMOS'
  MasterFields = 'CustNo'
  MasterSource = DataSourceCust
  TableName = 'ORDERS.DB'
end
object DataSourceCust: TDataSource
  DataSet = TableCustomer
end
object ProviderCustomer: TDataSetProvider
  DataSet = TableCustomer
end
```

On the client side, the detail table will show up as an extra field of the ClientDataSet, and the DBGrid control will display it as an extra column with an ellipsis button. Clicking the button will display a secondary form with a grid presenting the detail table (see Figure 17.6). If you need to build a flexible user interface on the client, you can then add a secondary ClientDataSet connected to the dataset field of the master dataset, using the DataSetField property. Simply create persistent fields for the main ClientDataSet and then hook up the property:

```
object cdsDet: TClientDataSet
  DataSetField = cdsTableOrders
end
```

With this setting you can show the detail dataset in a separate DBGrid placed as usual in the form (the bottom grid of Figure 17.6) or in any other way you like. Note that with this structure, the updates relate only to the master table, and the server should handle the proper update sequence even in complex situations.



marco

marco

☑ Connect

	TaxRate	Contact	LastInvoiceDate	TableOrders
▶	8.5	Erica Norman	2/2/95 1:05:03 AM	ATASET) ···
	0	George Weathers	11/17/94 2:10:33 PM	(DATASET)
	0	Phyllis Spooner	10/18/94 7:20:30 PM	(DATASET)
	0	Joe Bailey	1/30/92 2:00:56 AM	(DATASET)
	0	Chris Thomas	3/20/92 9:35:40 AM	(DATASET)
	0	Ernest Barratt	11/8/94 11:22:08 PM	(DATASET)
	0	Russell Christopher	2/1/95 6:45:23 PM	(DATASET)
	0	Paul Gardner	11/9/94 1:22:22 AM	(DATASET)

	OrderNo	CustNo	SaleDate
▶	1023	1221	7/1/88
	1076	1221	12/16/94
	1123	1221	8/24/93
	1169	1221	7/6/94
	1176	1221	7/26/94
	1269	1221	12/16/94

	OrderNo	CustNo	SaleDate	ShipDate	EmpNo
▶	1023	1221	7/1/88	7/2/88	5
	1076	1221	12/16/94	4/26/89	9
	1123	1221	8/24/93	8/24/93	121
	1169	1221	7/6/94	7/6/94	12
	1176	1221	7/26/94	7/26/94	52
	1269	1221	12/16/94	12/16/94	28

FIGURE 17.6:

The ThinPlus example shows how a dataset field can either be displayed in a grid in a floating window or extracted by a ClientDataSet and displayed in a second form. You'll generally do one of the two things, not both!

Using the Connection Broker

I've already mentioned that the ConnectionBroker component can be helpful in case you might want to change the physical connection used by many ClientDataSet components of a single program. In fact, by hooking each ClientDataSet to the ConnectionBroker, you can change the physical connection of them all simply by updating the physical connection of the broker.

These are the settings used by the ThinPlus example:

```
object Connection: TSocketConnection
  ServerName = 'AppSPlus.AppServerPlus'
  AfterConnect = ConnectionAfterConnect
  Address = '127.0.0.1'
end
object ConnectionBroker1: TConnectionBroker
  Connection = Connection
end
```

```
object cds: TClientDataSet
  ConnectionBroker = ConnectionBroker1
end
// in the secondary form
object cdsQuery: TClientDataSet
  ConnectionBroker = ClientForm.ConnectionBroker1
end
```

That's basically all you have to do. To change the physical connection, drop a new DataSnap connection component to the main form and set the `Connection` property of the broker to it.

WARNING There are some glitches with the ConnectionBroker, even in the shipping version of Delphi 6. If you experience unusual errors in a program that uses this component, try removing it. Of course, this note applies only until Borland provides a patch to fix this behavior.

More Provider Options

I've already mentioned the `Options` property of the DataSetProvider component, noting that it can be used to add the field properties to the data packet. There are several other options you can use to customize the data packet and the behavior of the client program. Here is a short list:

- You can minimize downloading BLOB data with poFetchBlobsOnDemand option. In this case, the client application can download BLOBs by specifying the `FetchOnDemand` property of the ClientDataSet to True or by calling the `FetchBlobs` method for specific records. Similarly, you can disable the automatic downloading of detail records by setting the poFetchDetailsOnDemand option. Again, the client can use the `FetchOnDemand` property or call the `FetchDetails` method.
- When you are using a master/detail relation, you can control cascades with either of two options. The poCascadeDeletes flag controls whether the provider should delete detail records before deleting a master record. You can set this option if the database server performs cascaded deletes for you as part of its referential integrity support. Similarly, you can set the poCascadeUpdates option when the update of key values of a master/detail relation can be performed automatically by the server.
- You can limit the operations on the client side. The most restrictive option, poReadOnly, disables any update. If you want to give the user a limited editing capability, use poDisableInserts, poDisableEdits, or poDisableDeletes.
- You can resend to the client a copy of the records the client has modified with `poAutoRefresh`, which is useful in case other users have simultaneously made other, nonconflicting changes. You can also send back to the client changes done in the `BeforeUpdateRecord`

or AfterUpdateRecord event handlers by specifying the poPropogateChanges option. This option is also handy when you are using autoincrement fields, triggers, and other techniques that modify data on the server or middle tier beyond the changes requested from the client tier.

- Finally, if you want the client to drive the operations, you can enable the poAllowCommandText option. This lets you set the SQL query or table name of the middle tier from the client, using the GetRecords or Execute methods.

The Simple Object Broker

The SimpleObjectBroker component provides an easy way to locate a server application among several server computers. You simply provide a list of available computers, and the client will try each of them in order until it finds one that is available.

Moreover, if you enable the LoadBalanced property, the component will randomly choose one of the servers; when many clients use the same configuration, the connections will be automatically distributed among the multiple servers. If this seems like a "poor man's" object broker, consider that some highly expensive load-balancing systems don't actually offer much more than this.

Object Pooling

When multiple clients connect to your server at the same time, you have two options. The first is to create a remote data module object for each of them and let each request be processed in sequence (the default behavior for a COM server with the ciMultiInstance style). Alternatively, you can let the system create a different instance of the application for every client (ciSingleInstance). This requires more resources and more SQL server connections (and licenses), potentially overloading the BDE (as it cannot handle more than a set number of threads or processes).

An alternative approach is offered by the support in DataSnap for object pooling. All you need to do to request this feature is add a call to RegisterPooled in the overridden UpdateRegistry method. Combined with the stateless support built into this architecture, the pooling capability allows you to share some middle-tier objects among a much larger number of clients.

The users on the client computers will spend most of their time reading data and typing in updates, and they generally don't continue asking for data and sending updates. When the client is not calling a method of the middle-tier object, this can be used for another client. Being stateless, in fact, every request reaches the middle tier as a brand-new operation, even when a server is dedicated to a specific client.

Pooling mechanisms are built into MTS and CORBA, but DataSnap makes it available also for HTTP and socket-based connections, and for the Internet Express Web client.

Customizing the Data Packets

There are many ways to include custom information within the data packet handled by the `IAppServer` interface. The simplest is probably to handle the `OnGetDataSetProperties` event of the provider itself. This event has a `Sender` parameter, a dataset parameter indicating where the data is coming from, and an `OleVariant` array `Properties` parameter, in which you can place the extra information. You need to define one variant array for each extra property and include the name of the extra property, its value, and whether you want the data to return to the server along with the update delta (the `IncludeInDelta` parameter).

Of course, you can pass properties of the related dataset component, but you can also pass any other value (extra fake properties). In the AppSPlus example, I pass to the client the time the query was executed and its parameters:

```
procedure TAppServerPlus.ProviderQueryGetDataSetProperties(
  Sender: TObject; DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['Time', Now, True]);
  Properties[1] := VarArrayOf(['Param', Query.Params[0].AsString, False]);
end;
```

On the client side, the ClientDataSet component has a `GetOptionalParameter` method to retrieve the value of the extra property with the given name. The ClientDataSet also has the `SetOptionalParameter` method to add more properties to the dataset. These values will be saved to disk (in the briefcase model) and eventually sent back to the middle tier (by setting the `IncludeInDelta` member of the variant array to True). Here is a simple example of the retrieval of the dataset in the code above:

```
Caption := 'Data sent at ' + TimeToStr (TDateTime (
  cdsQuery.GetOptionalParam('Time')));
Label1.Caption := 'Param ' + cdsQuery.GetOptionalParam('Param');
```

The effect of this code was visible in Figure 17.5. An alternative and more powerful approach for customizing the data packet sent to the client is to handle the `OnGetData` event of the provider, which receives the outgoing data packet in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client. For example, you might encode some of the data or filter out sensitive records.

What's Next?

Borland originally introduced its multitier technology in Delphi 3 and has kept extending it from version to version. In addition to further updates and the change of the MIDAS name to DataSnap, Delphi 6 sees the introduction of XML and SOAP support, introducing an alternate and extended architecture for multitier applications. We'll fully explore this topic in Chapter 23.

For the moment, we'll continue with database programming, discussing data-aware controls and custom datasets. In the next part of the book we'll explore COM, sockets, and Internet programming, getting to XML and SOAP at the end of the book, after we've discussed a lot of foundation material.

CHAPTER 18

Writing Database Components

- Data-aware components: the data link
- Field-oriented data-aware controls
- Data-aware TrackBar and ProgressBar
- Record-oriented data-aware controls
- A record viewer
- Building custom datasets
- Saving a dataset to a local stream

In Chapter 11, “Creating Components,” we explored the development of Delphi components in depth. Now that I’ve discussed database programming, we can get back to the earlier topic and focus on the development of database-related components.

There are basically two families of such components. There are data-aware controls you can use to present the data of a field or an entire record to the users of a program. There are dataset components you can define to provide data to existing data-aware controls, reading it from a database or any other data source. In this chapter, I’ll cover both topics.

The Data Link

When you write a Delphi database program, you generally connect some data-aware controls to a DataSource component, and then connect the DataSource component to a dataset. The connection between the data-aware control to the DataSource is called a *data link* and is represented by an object of class `TDataLink`. The data-aware control creates and manages this object and represents its only connection to the data. From a more practical perspective, to make a component data-aware, you need to add a data link to it and surface some of the properties of this internal object, such as the `DataSource` and `DataField` properties.

Delphi uses the DataSource and DataLink objects for bidirectional communication. The dataset uses the connection to notify the data-aware controls that new data is available (because the dataset has been activated, or the current record has changed, and so on). Data-aware controls use the connection to ask for the current value of a field or to update it, notifying the dataset of this event.

The relations among all these components are complicated by the fact that some of the connections can be one-to-many. For example, you can connect multiple data sources to the same dataset, and you generally have multiple data links to the same data source, simply because you need one link for every data-aware component, and in most cases you connect multiple data-aware controls to each data source.

The *TDataLink* Class

We’ll work for much of this chapter with `TDataLink` and its derived classes, which are defined in the DB unit. This class has a set of protected virtual methods, which have a role similar to events. They are “almost-do-nothing” methods you can override in a specific subclass to intercept user operations and other data-source events. Here is a list, extracted from the source code of the class:

```
type
  TDataLink = class(TPersistent)
  protected
    procedure ActiveChanged; virtual;
    procedure CheckBrowseMode; virtual;
    procedure DataSetChanged; virtual;
    procedure DataSetScrolled(Distance: Integer); virtual;
    procedure FocusControl(Field: TFieldRef); virtual;
    procedure EditingChanged; virtual;
    procedure LayoutChanged; virtual;
    procedure RecordChanged(Field: TField); virtual;
    procedure UpdateData; virtual;
```

All of these virtual methods are called by the DataEvent private method, a sort of window procedure for a data source, a procedure triggered by several data events (see the TDataEvent enumeration). These events originate in the dataset, fields, or data source, and are generally applied to a dataset. The DataEvent method of the dataset component dispatches the events to the connected data sources. Each data source calls the NotifyDataLinks method to forward the event to each connected data link, and then the data source triggers either its own OnDataChange or OnUpdateData event.

Derived DataLink Classes

The TDataLink class is not technically an abstract class, but you'll seldom use it directly. When you need to create data-aware controls, you'll need to use one of its derived classes or derive a new one yourself. The most important class derived from TDataLink is the TFieldDataLink class, which is used by data-aware controls that relate to a single field of the dataset. Most data-aware controls fall into this category, and the TFieldDataLink class solves the most common problems of this type of component.

All of the table- or record-oriented data-aware controls define specific subclasses of TDataLink, as we'll do later on. The TFieldDataLink class has a list of events corresponding to the virtual methods of the base class it overrides. This makes the class simpler to customize, as you can use event handlers instead of having to inherit a new class from it. Here's an example of an overridden method, which fires the corresponding event, if available:

```
procedure TFieldDataLink.ActiveChanged;
begin
  UpdateField;
  if Assigned(FOnActiveChange) then FOnActiveChange(Self);
end;
```

The TFieldDataLink class contains also the Field and FieldName properties that let you connect the data-aware control to a specific field of the dataset. The link keeps also a reference to the current visual component, using the Control property.

Writing Field-Oriented Data-Aware Controls

Now that you understand the theory of how the data link classes work, I can start building some data-aware controls. The first two examples I'll build are data-aware versions of the ProgressBar and TrackBar common controls. We can use the first to display a numeric value, such as a percentage, in a visual way. We can use the second to allow a user to change the numeric value as well.

A Read-Only ProgressBar

A data-aware version of the ProgressBar control is a relatively simple case of a data-aware control, because it is a read-only control. This component is derived from the version that's not data-aware and adds a few properties of the data link object it encapsulates:

```
type
  TMdDbProgress = class(TProgressBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
  protected
    // data link event handler
    procedure DataChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

As with every data-aware component that connects to a single field, this control makes available the DataSource and DataField properties. There is very little code to write here; simply export the properties from the internal data link object, as follows:

```
function TMdDbProgress.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TMdDbProgress.SetDataField (Value: string);
begin
```

```
  FDataLink.FieldName := Value;
end;

function TMdDbProgress.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TMdDbProgress.SetDataSource (Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

function TMdDbProgress.GetField: TField;
begin
  Result := FDataLink.Field;
end;
```

Of course, to make this component work, you must create and destroy the data link when the component itself is created or destroyed:

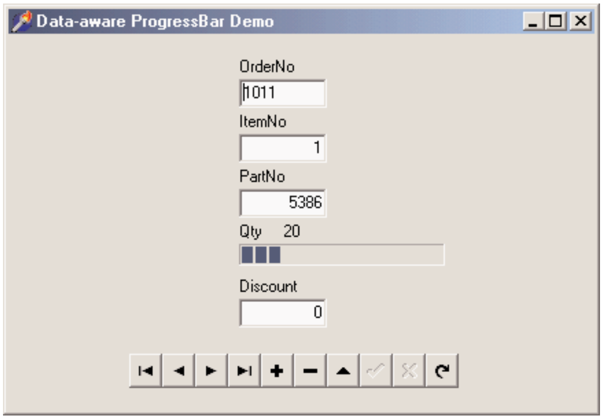
```
constructor TMdDbProgress.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := self;
  FDataLink.OnDataChange := DataChange;
end;

destructor TMdDbProgress.Destroy;
begin
  FDataLink.Free;
  FDataLink := nil;
  inherited Destroy;
end;
```

In the preceding constructor, notice that the component installs one of its own methods as an event handler for the data link. This is where the most important code of the component resides. Every time the data changes, we modify the output of the progress bar to reflect the values of the current field:

```
procedure TMdDbProgress.DataChange (Sender: TObject);
begin
  if (FDataLink.Field <> nil) and (FDataLink.Field is TNumericField) then
    Position := FDataLink.Field.AsInteger
  else
    Position := Min;
end;
```

Following the convention of the VCL data-aware controls, if the field type is invalid, the component doesn't display an error message—it simply disables the output. Alternatively, you might want to check the field type when `SetDataField` method assigns it to the control.

In Figure 18.1 you can see an example of the DbProgr application's output, which uses both a label and a progress bar to display an order's quantity information. Thanks to this visual clue, you can step through the records and easily spot orders for many items. One obvious benefit to this component is that the application contains almost no code, since all the important code is in the component itself.



Data-aware ProgressBar Demo

OrderNo 1011

ItemNo 1

PartNo 5386

Qty 20

Discount 0

FIGURE 18.1: The data-aware ProgressBar in action in the DbProgr example

As you've seen, a read-only data-aware component is not too difficult to write. It gets extremely complex, on the other hand, to use such a component inside a DBCtrlGrid container.

NOTE If you remember the discussion of the `Notification` method in Chapter 11, you might wonder what happens if the data source referenced by the data-aware control is destroyed. The good news is that the data source has a destructor that removes itself from its own data links. So there is no need for a `Notification` method for data-aware controls, even though you'll see books and articles suggesting it, and VCL has plenty of this extra useless code.

Replicable Data-Aware Controls

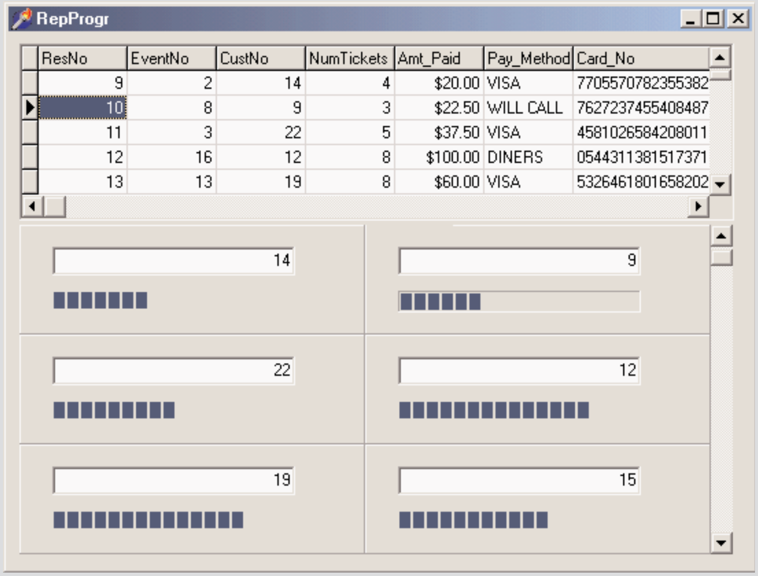
Extending a data-aware control to support its use inside a DBCtrlGrid component is rather complex and not well documented. You can find a complete "replicable" version of the progress bar in the MdDataPack package and an example of its use in the `RepProgr` folder, along with an HTML file describing its development. The DBCtrlGrid component has a peculiar behavior, as it displays on screen multiple versions of the same physical control, using some

Continued on next page

"smoke and mirrors." The grid can attach the control to a data buffer other than the current record and redirects the control paint operations to another portion of the monitor.

In short, to appear in the DBCtrlGrid, a component must have its csReplicatable control style set, a flag merely indicating that your component actually supports being hosted by a control grid. First, the component must respond to the `cm_GetDataLink` Delphi message and return a pointer to the data link, so that the control grid can use and change it. Second, it needs a custom `Paint` method to draw the output in the appropriate canvas object, which is provided in a parameter of the `wm_Paint` message in case the csPaintCopy flag of the `ControlState` property is set.

The actual code of the example is rather complex, and the DBCtrlGrid component is not heavily used, so I decided not to give you full details here, but you can find the full code and some more information in the source code on the companion CD. Here's the output of a test program that uses this component:



ResNo	EventNo	CustNo	NumTickets	Amt_Paid	Pay_Method	Card_No
9	2	14	4	$20.00	VISA	7705570782355382
10	8	9	3	$22.50	WILL CALL	7627237455408487
11	3	22	5	$37.50	VISA	4581026584208011
12	16	12	8	$100.00	DINERS	0544311381517371
13	13	19	8	$60.00	VISA	5326461801658202

A Read-Write TrackBar

The next step is to write a component that allows a user to modify the data in a database, not just browse it. The overall structure of this type of component isn't very different from the previous version, but there are a few extra elements. In particular, when the user starts interacting with the component, the code should put the dataset into edit mode and then notify the dataset that the data has changed. The dataset will then use an event handler of the FieldDataLink to ask for the updated value.

To demonstrate how you can create a data-aware component that modifies the data, I've decided to extend the TrackBar control. This probably isn't the simplest example, but it demonstrates several important techniques.

Here's the definition of the component's class:

```
type
  TMdDbTrack = class(TTrackBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
    procedure CMExit(var Message: TCMExit); message CM_EXIT;
  protected
    // data link event handlers
    procedure DataChange (Sender: TObject);
    procedure UpdateData (Sender: TObject);
    procedure ActiveChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

Compared to the read-only data-aware control built earlier, this class is a bit more complex, because it has three message handlers, including component notification handlers, and two new event handlers for the data link. The component installs these event handlers in the constructor, which also disables the component:

```
constructor TMdDbTrack.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := self;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
  FDataLink.OnActiveChange := ActiveChange;
  Enabled := False;
end;
```

All of the get and set methods and the DataChange event handler are very similar to those in the TMdDbProgress component. The only difference is that whenever the data source or data field changes, the component checks the current status to see whether it should enable itself:

```
procedure TMdDbTrack.SetDataSource (Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;
```

This code tests three conditions: the data link should be active, the link should refer to an actual field, and the field shouldn't be read-only. When the user changes the field, the component should also consider that the field name might be invalid; to test for this condition, the component should rather use a try/finally block:

```
procedure TMdDbTrack.SetDataField (Value: string);
begin
  try
    FDataLink.FieldName := Value;
  finally
    Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
      not FDataLink.Field.ReadOnly;
  end;
end;
```

The control executes the same test when the dataset is enabled or disabled:

```
procedure TMdDbTrack.ActiveChange (Sender: TObject);
begin
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;
```

The most interesting portion of this component's code is related to its user interface. When a user starts moving the scroll thumb, the component should do the following: put the dataset into edit mode, let the base class update the thumb position, and alert the data link (and therefore the data source) that the data has changed. Here's the code:

```
procedure TMdDbTrack.CNHScroll(var Message: TWMHScroll);
begin
  // enter edit mode
  FDataLink.Edit;
  // update data
  inherited;
  // let the system know
  FDataLink.Modified;
end;
```

```
procedure TMdDbTrack.CNVScroll(var Message: TWMVScroll);
begin
  // enter edit mode
  FDataLink.Edit;
  // update data
  inherited;
  // let the system know
  FDataLink.Modified;
end;
```

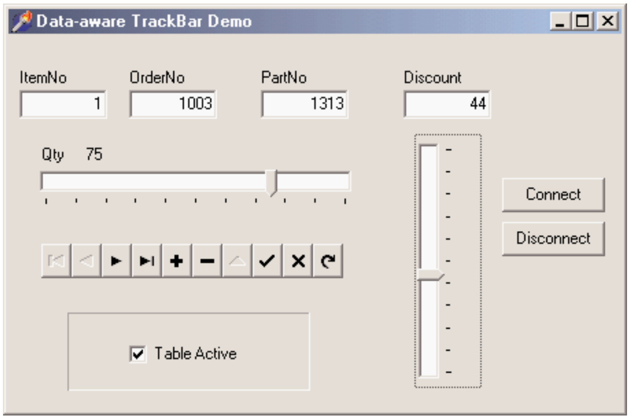
When the dataset needs new data—for example, to perform a `Post` operation—it simply requests it from the component via the `TFieldDataLink` class's `OnUpdateData` event:

```
procedure TMdDbTrack.UpdateData (Sender: TObject);
begin
  if (FDataLink.Field <> nil) and (FDataLink.Field is TNumericField) then
    FDataLink.Field.AsInteger := Position;
end;
```

If the proper conditions are met, the component simply updates the data in the proper table field. Finally, if the component loses the input focus, it should force a data update (if the data has changed) so that any other data-aware components showing the value of that field will display the correct value as soon as the user moves to a different field. If the data hasn't changed, the component won't bother updating the data in the table. This is the standard `CmExit` code for components used by VCL and borrowed for our component as well:

```
procedure TMdDbTrack.CmExit(var Message: TCmExit);
begin
  try
    FDataLink.UpdateRecord;
  except
    SetFocus;
    raise;
  end;
  inherited;
end;
```

Again, there is a demo program for testing this component; you can see its output in Figure 18.2. The DbTrack program contains a check box to enable and disable the table, the visual components, and a couple of buttons you can use to detach the vertical TrackBar component from the field it relates to. Again, I placed these on the form to test enabling and disabling the track bar.



Data-aware TrackBar Demo

ItemNo: 1

OrderNo: 1003

PartNo: 1313

Discount: 44

Qty 75

☑ Table Active

FIGURE 18.2: The DbTrack example has a couple of track bars you can use to enter data in a database table. The check box and buttons are used to test the enabled status of the components.

Creating Custom Data Links

The data-aware controls I've built up to this point all referred to specific fields of the dataset, so I was able to use a `TFieldDataLink` object to establish the connection with a data source. Now I want to build a data-aware component that works with a dataset as a whole, a simple record viewer.

Delphi's database grid shows the value of several fields and several records simultaneously. In my record viewer component, I want to list all the fields of the current record, using a customized grid. This example will show you how to build a customized grid control, and a custom data link to go with it.

A Record Viewer Component

In Delphi there are no data-aware components that manipulate multiple fields of a single record, without displaying other records. In fact, the only component that displays multiple fields from the same table is the DBGrid, which displays multiple fields and multiple records.

The record viewer component I'm going to describe in this section is based on a two-column grid; the first column displays the table's field names, while the second column displays the corresponding field values. The number of rows in the grid will correspond to the number of fields, with a vertical scroll bar in case they can't fit in the visible area.

The data link we need in order to build this component is a simple class, connected only to the record viewer component, and declared directly in the implementation portion of its unit. This is the same approach used by VCL for some specific data links. Here's the definition of the new class:

```
type
  TMdRecordLink = class (TDataLink)
  private
    RView: TMdRecordView;
  public
    constructor Create (View: TMdRecordView);
    procedure ActiveChanged; override;
    procedure RecordChanged (Field: TField); override;
  end;
```

As you can see, the class overrides the methods related to the principal event, in this case simply the activation and data (or record) change. Alternatively, I could have exported some events and then let the component handle them. That's what the TFieldDataLink does, but the approach I've taken makes more sense for a data link class, because you'll want to use it with different data-aware components. The constructor requires the associated component as its only parameter:

```
constructor TMdRecordLink.Create (View: TMdRecordView);
begin
  inherited Create;
  RView := View;
end;
```

After storing a reference to the associated component, the other methods can operate on it directly:

```
procedure TMdRecordLink.ActiveChanged;
var
  I: Integer;
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // repaint all...
  RView.Invalidate;
end;

procedure TMdRecordLink.RecordChanged;
begin
  inherited;
  // repaint all...
  RView.Invalidate;
end;
```

As you've seen, the record link code is very simple. Most of the difficulties in building this example depend on the use of a grid. To avoid dealing with useless properties, I've derived the record viewer grid from the TCustomGrid class. This class incorporates much of the code for grids, but most of its properties, events, and methods are protected. For this reason, the class declaration is quite long, because it needs to publish many existing properties. Here is an excerpt (excluding the base class properties):

```
type
  TMdRecordView = class(TCustomGrid)
  private
    // data-aware support
    FDataLink: TDataLink;
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
  protected
    // redefined TCustomGrid methods
    procedure DrawCell (ACol, ARow: Longint; ARect: TRect;
      AState: TGridDrawState); override;
    procedure ColWidthsChanged; override;
    procedure RowHeightsChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure SetBounds (ALeft, ATop, AWidth, AHeight: Integer); override;
    procedure DefineProperties (Filer: TFiler); override;
    // public parent properties (omitted...)
  published
    // data-aware properties
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    // published parent properties (omitted...)
  end;
```

Besides redeclaring the properties to publish them, the component defines a data link object and the `DataSource` property. There's no `DataField` property for this component, because it refers to an entire record. The component's constructor is very important. It sets the values of many unpublished properties, including the grid options:

```
constructor TMdRecordView.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TMdRecordLink.Create (self);
  // set numbers of cells and fixed cells
  RowCount := 2; // default
  ColCount := 2;
  FixedCols := 1;
  FixedRows := 0;
  Options := [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goRowSizing];
  DefaultDrawing := False;
  ScrollBars := ssVertical;
  FSaveCellExtents := False;
end;
```

The grid has two columns, one of them fixed, and no fixed rows. The fixed column is used for resizing each row of the grid. Unfortunately, a user cannot drag the fixed row to resize the columns, because you can't resize fixed elements, and the grid already has a fixed column.

NOTE An alternative approach could be to have an extra empty column, as the DBGrid control does. You'd be able to resize the two other columns after adding a fixed row. Overall, though, I prefer my implementation.

I've used an alternative approach to resize the columns. The first column (holding the field names) can be resized either using programming code or visually at design time, and the second column (holding the values of the fields) will be resized to use the remaining area of the component, leaving space for the borders, lines, and vertical scrollbar:

```
procedure TMdRecordView.SetBounds (ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited;
  ColWidths [1] := Width - ColWidths [0] - GridLineWidth * 3 -
    GetSystemMetrics (sm_CXVScroll) - 2; // border
end;
```

This takes place when the component size changes and when either of the columns change. With this code, the `DefaultColWidth` property of the component becomes, in practice, the fixed width of the first column.

After everything has been set up, the key method of the component is the overridden `DrawCell` method, detailed in Listing 18.1. This is where the control displays the information about the fields and their values. There are three things it needs to draw. If the data link is not connected to a data source, the grid displays an "empty element" sign (*[]*). When drawing the first column, the record viewer shows the `DisplayName` of the field, which is the same value used by the DBGrid for the heading. When drawing the second column, the component accesses the textual representation of the field value, extracted with the `DisplayText` property (or with the `AsString` property for memo fields).

Listing 18.1: The `DrawCell` method of the custom RecordView component

```
procedure TMdRecordView.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
var
  Text: string;
  CurrField: TField;
  Bmp: TBitmap;
begin
  CurrField := nil;
  Text := '[]'; // default
  // paint background
```

```
    if (ACol = 0) then
      Canvas.Brush.Color := FixedColor
    else
      Canvas.Brush.Color := Color;
    Canvas.FillRect (ARect);
    // leave small border
    InflateRect (ARect, -2, -2);
    if (FDataLink.DataSource <> nil) and FDataLink.Active then
    begin
      CurrField := FDataLink.DataSet.Fields[ARow];
      if ACol = 0 then
        Text := CurrField.DisplayName
      else if CurrField is TMemoField then
        Text := TMemoField (CurrField).AsString
      else
        Text := CurrField.DisplayText;
    end;
    if (ACol = 1) and (CurrField is TGraphicField) then
    begin
      Bmp := TBitmap.Create;
      try
        Bmp.Assign (CurrField);
        Canvas.StretchDraw (ARect, Bmp);
      finally
        Bmp.Free;
      end;
    end
    else if (ACol = 1) and (CurrField is TMemoField) then
    begin
      DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_WordBreak or dt_NoPrefix)
    end
    else // draw single line vertically centered
      DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_vcenter or dt_SingleLine or dt_NoPrefix);
    if gdFocused in AState then
      Canvas.DrawFocusRect (ARect);
  end;
```

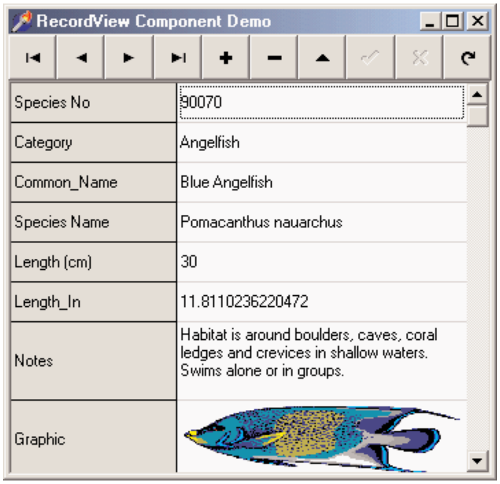
The final portion of the method is where the component considers memo and graphic fields. If the field is a TMemoField, the DrawText function call doesn't specify the dt_SingleLine flag, but uses dt_WordBreak flag to wrap the words when there's no more room. For a graphic field, of course, the component uses a completely different approach, assigning the field image to a temporary bitmap, and then stretching it to fill the surface of the cell.

Notice also that the component sets the DefaultDrawing property to False, so that it's also responsible for drawing the background and the focus rectangle, as it does in the DrawCell method. The component also calls the InflateRect API function to leave a small area between the cell border and the output text. The actual output is produced by calling another Windows API function, `DrawText`, which centers the text vertically in its cell.

This drawing code works both at run time, as you can see in Figure 18.3, and at design time. The output may not be perfect, but this component can certainly be very useful in many cases. If you want to display the data for a single record, instead of building a custom form with labels and data-aware controls, you can easily use this record viewer grid. Of course, it's important to remember that the record viewer is a read-only component: it's certainly possible to extend it to add editing capabilities (they're already part of the `TCustomGrid` class). However, instead of adding this support, we've decided to make the component more complete by adding support for displaying BLOB fields.

FIGURE 18.3:

The ViewGrid example demonstrates the output of the RecordView component, using Borland's sample BioLife database table.



Species No	90070
Category	Angelfish
Common_Name	Blue Angelfish
Species Name	Pomacanthus nauarchus
Length (cm)	30
Length_In	11.8110236220472
Notes	Habitat is around boulders, caves, coral ledges and crevices in shallow waters. Swims alone or in groups.
Graphic	

To improve the graphical output, the control makes the lines for those fields twice as high as those for plain text fields. This operation is accomplished when the dataset connected to the data-aware control is activated. The `ActiveChanged` method of the data link is triggered also by the `RowHeightsChanged` methods, connected to the `DefaultRowHeight` property of the base class:

```
procedure TMdRecordLink.ActiveChanged;
var
  I: Integer;
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // double the height of memo and graphics
  for I := 0 to DataSet.FieldCount - 1 do
    if DataSet.Fields [I] is TBlobField then
```

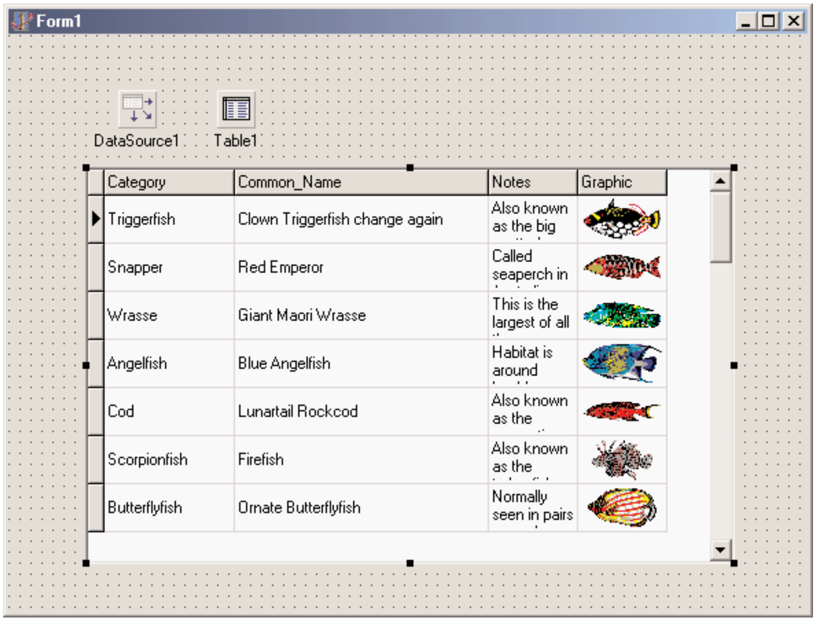
```
        RView.RowHeights [I] := RView.DefaultRowHeight * 2;
    // repaint all...
    RView.Invalidate;
  end;
```

At this point, we stumble into a minor problem. In the `DefineProperties` method, the `TCustomGrid` class saves the values of the `RowHeights` and `ColHeights` properties. We could disable this streaming by overriding the method and not calling `inherited` (which is generally a bad technique to use), but it is also possible to toggle the `FSaveCellExtents` protected field to disable this feature.

Customizing the DBGrid Component

Besides writing brand-new custom data-aware components, it's common for Delphi programmers to customize the DBGrid control. The goal for the next component is to enhance the DBGrid with the same kind of custom output I've used for the RecordView component, directly displaying graphic and memo fields. To do this, the grid needs to make the row height resizable, to allow space for a reasonable amount of text and big enough for graphics. You can see an example of this grid at design time in Figure 18.4.

FIGURE 18.4: An example of the MdDbGrid component at design time. Notice the output of the graphics and memo fields.



Category	Common_Name	Notes	Graphic
Triggerfish	Clown Triggerfish change again	Also known as the big	
Snapper	Red Emperor	Called seaperch in	
Wrasse	Giant Maori Wrasse	This is the largest of all	
Angelfish	Blue Angelfish	Habitat is around	
Cod	Lunartail Rockcod	Also known as the	
Scorpionfish	Firefish	Also known as the	
Butterflyfish	Ornate Butterflyfish	Normally seen in pairs	

While creating the output was a simple matter of adapting the code used in the record viewer component, setting the height of the grid cells ended up being a very difficult problem to solve. The lines of code you'll see for that operation may be few, but they cost me hours of work!

NOTE Unlike the generic grid we've used above, a DBGrid is a virtual view on the dataset—there is no relation between the number of rows shown on the screen and the number of rows of data in the dataset. When you scroll up and down through the data records of the dataset, you are not scrolling up and down through the rows of the DBGrid; the rows are stationary while the data moves from one row to the next to give the appearance of movement. For this reason, the program doesn't try to set the height of an individual row to suit its data, but it sets the height of all the data rows to a multiline height value.

This time the control doesn't have to create a custom data link, because it is deriving from a component that already has a complex connection with the data. The new class has a new property to specify the number of lines of text for each row and overrides a few virtual methods:

```
type
  TMdDbGrid = class(TDbGrid)
  private
    FLinesPerRow: Integer;
    procedure SetLinesPerRow (Value: Integer);
  protected
    procedure DrawColumnCell(const Rect: TRect; DataCol: Integer;
      Column: TColumn; State: TGridDrawState); override;
    procedure  LayoutChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
  published
    property LinesPerRow: Integer
      read FLinesPerRow write SetLinesPerRow default 1;
  end;
```

The constructor simply sets the default value for the `FLinesPerRow` field. Here is the set method for the property:

```
procedure TMdDbGrid.SetLinesPerRow(Value: Integer);
begin
  if Value <> FLinesPerRow then
  begin
    FLinesPerRow := Value;
    LayoutChanged;
  end;
end;
```

The side effect of changing the number of lines is a call to the LayoutChanged virtual method. The system calls this method frequently when one of the many output parameters changes. In the code of this method, the component first calls the inherited version and then sets the height of each row. As a basis for this computation it uses the same formula of the TCustomDBGrid class: the text height is calculated using the sample word *Wg* in the current font (this text is used because it includes both a full-height uppercase character and a lowercase letter with a descender). Here's the code:

```
procedure TMdDbGrid.LayOutChanged;
var
  PixelsPerRow, PixelsTitle, I: Integer;
begin
  inherited LayOutChanged;

  Canvas.Font := Font;
  PixelsPerRow := Canvas.TextHeight('Wg') + 3;
  if dgRowLines in Options then
    Inc (PixelsPerRow, GridLineWidth);

  Canvas.Font := TitleFont;
  PixelsTitle := Canvas.TextHeight('Wg') + 4;
  if dgRowLines in Options then
    Inc (PixelsTitle, GridLineWidth);

  // set number of rows
  RowCount := 1 + (Height - PixelsTitle) div (PixelsPerRow * FLinesPerRow);

  // set the height of each row
  DefaultRowHeight := PixelsPerRow * FLinesPerRow;
  RowHeights [0] := PixelsTitle;
  for I := 1 to RowCount - 1 do
    RowHeights [I] := PixelsPerRow * FLinesPerRow;
end;
```

WARNING Font and TitleFont are the grid defaults that can be overridden by properties of the individual DBGrid column objects. This component would currently ignore those settings.

The difficult part here was to get the last four statements correct. You can simply set the DefaultRowHeight property, but in that case the title row will probably be too high. At first, I tried setting the DefaultRowHeight and then the height of the first row, but this complicated the code used to compute the number of visible rows in the grid (the read-only VisibleRowCount property). If you specify the number of rows (in order to avoid having rows hidden beneath the lower edge of the grid), the base class keeps recomputing them.

Finally, here's the code used to draw the data, ported from the RecordView component and adapted slightly for the grid:

```
procedure TMdDbGrid.DrawColumnCell (const Rect: TRect; DataCol: Integer;
  Column: TColumn; State: TGridDrawState);
var
  Bmp: TBitmap;
  OutRect: TRect;
begin
  if FLinesPerRow = 1 then
    inherited DrawColumnCell(Rect, DataCol, Column, State)
  else
  begin
    // clear area
    Canvas.FillRect (Rect);
    // copy the rectangle
    OutRect := Rect;
    // restrict output
    InflateRect (OutRect, -2, -2);
    // output field data
    if Column.Field is TGraphicField then
    begin
      Bmp := TBitmap.Create;
      try
        Bmp.Assign (Column.Field);
        Canvas.StretchDraw (OutRect, Bmp);
      finally
        Bmp.Free;
      end;
    end
    else if Column.Field is TMemoField then
    begin
      DrawText (Canvas.Handle, PChar (Column.Field.AsString),
        Length (Column.Field.AsString), OutRect, dt_WordBreak or dt_NoPrefix)
    end
    else // draw single line vertically centered
      DrawText (Canvas.Handle, PChar (Column.Field.DisplayText),
        Length (Column.Field.DisplayText), OutRect,
        dt_vcenter or dt_SingleLine or dt_NoPrefix);
  end;
end;
```

In the code above you can see that if the user displays just a single line, the grid uses the standard drawing technique with no output for memo and graphic fields. However, as soon as you increase the line count, you'll see a better output.

To see this code in action, run the GridDemo example. This program has two buttons you can use to increase or decrease the row height of the grid, and two more buttons to change the font. This is an important test because the height in pixels of each cell is the height of the font multiplied by the number of lines.

Building Custom Datasets

When discussing the TDataSet class and the alternative families of dataset components available in Delphi, in Chapter 13, "Delphi's Database Architecture," I mentioned the possibility of writing a custom dataset class. Now it's time to have a look at an actual example. The reasons for writing a custom dataset relate to the fact that you won't need to deploy a database engine but you'll still be able to take full advantage of Delphi's database architecture, including things like persistent database fields and data-aware controls.

Writing a custom dataset is one of the most complex task for a component developer, so this is one of the most advanced areas (as far as low-level coding practices, including tons of pointers) of the entire book. Moreover, Borland hasn't released any official documentation on writing custom datasets. If you are early in your experience with Delphi, you might want to skip the rest of this chapter and come back here later.

The TDataSet class is an abstract class, which declares several virtual abstract methods—23 to be precise. Every subclass of TDataSet must override all of those methods.

Before discussing the development of a custom dataset, we need to explore a few technical elements of the TDataSet class, in particular record buffering. The class maintains a list of buffers, which store the values of different records. These buffers store the actual data, but they also usually store further information for the dataset to use when managing the records. These buffers don't have a predefined structure, and each custom dataset must allocate the buffers, fill them, and destroy them. The custom dataset must also copy the data from the record buffers to the various fields of the dataset, and vice versa. In other words, the custom dataset is entirely responsible for handling these buffers.

In addition to managing the data buffers, the component is also responsible for navigating among the records, managing the bookmarks, defining the structure of the dataset, and creating the proper data fields. The TDataSet class is nothing more than a framework; you must fill it with the appropriate code. Fortunately, most of the code follows a standard structure, which the TDataSet-derived VCL classes use. Once you've grasped the key ideas, you'll be able to build multiple custom datasets borrowing quite a lot of code.

To simplify this type of reuse, I've collected the common features required by any custom dataset in a TMDCustomDataSet class. However, I'm not going to discuss the base class first and the specific implementation later, because that would probably be rather complex to understand. Instead, I'll detail the code required by a dataset, presenting methods of the generic and specific classes at the same time, according to a logical flow.

The Definition of the Classes

The starting point, as usual, is the declaration of the two classes discussed in this section, the generic custom dataset I've written and the specific component storing data in a stream. These declaration of these classes is available in Listing 18.2. Besides virtual methods, the classes contain a series of protected fields used to manage the buffers, track the current position and record count, and handle many other features. You should also notice another record declaration at the beginning, a structure used to store the extra data for every data record we place in a buffer. The dataset places this information in each record buffer, following the actual data.

Listing 18.2: The declaration of `TMdCustomDataSet` and `TMdDataSetStream`

```
// in the unit MdDsCustom
type
  EMdDataSetError = class (Exception);

  TMdRecInfo = record
    Bookmark: Longint;
    BookmarkFlag: TBookmarkFlag;
  end;
  PMdRecInfo = ^TMdRecInfo;

  TMdCustomDataSet = class(TDataSet)
  protected
    // status
    FIsTableOpen: Boolean;
    // record data
    FRecordSize,        // the size of the actual data
    FRecordBufferSize,  // data + housekeeping (TRecInfo)
    FCurrentRecord,     // current record (0 to FRecordCount - 1)
    BofCrack,           // before the first record (crack)
    EofCrack: Integer;  // after the last record (crack)
    // create, close, and so on
    procedure InternalOpen; override;
    procedure InternalClose; override;
    function IsCursorOpen: Boolean; override;
    // custom functions
    function InternalRecordCount: Integer; virtual; abstract;
    procedure InternalPreOpen; virtual;
    procedure InternalAfterOpen; virtual;
    procedure InternalLoadCurrentRecord(Buffer: PChar); virtual; abstract;
    // memory management
    function AllocRecordBuffer: PChar; override;
    procedure InternalInitRecord(Buffer: PChar); override;
    procedure FreeRecordBuffer(var Buffer: PChar); override;
    function GetRecordSize: Word; override;
    // movement and optional navigation (used by grids)
    function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):
      TGetResult; override;
    procedure InternalFirst; override;
    procedure InternalLast; override;
    function GetRecNo: Longint; override;
    function GetRecordCount: Longint; override;
    procedure SetRecNo(Value: Integer); override;
    // bookmarks
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    // editing (dummy vesions)
    procedure InternalDelete; override;
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
    procedure InternalPost; override;
    procedure InternalInsert; override;
    // other
    procedure InternalHandleException; override;
  published
    // redeclared dataset properties
    property Active;
    property BeforeOpen;
    property AfterOpen;
    property BeforeClose;
    property AfterClose;
    property BeforeInsert;
    property AfterInsert;
    property BeforeEdit;
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnCalcFields;
    property OnDeleteError;
    property OnEditError;
    property OnFilterRecord;
    property OnNewRecord;
    property OnPostError;
  end;

// in the unit MdDsStream
type
  TMdDataFileHeader = record
    VersionNumber: Integer;
    RecordSize: Integer;
    RecordCount: Integer;
  end;

  TMdDataSetStream = class(TMdCustomDataSet)
  private
    procedure SetTableName(const Value: string);
  protected
    FDataFileHeader: TMdDataFileHeader;
    FDataFileHeaderSize,    // optional file header size
    FRecordCount: Integer;  // current number of records
    FStream: TStream;       // the physical table
    FTableName: string;     // table path and file name
    FFieldOffset: TList;    // field offsets in the buffer
  protected
    // open and close
    procedure InternalPreOpen; override;
    procedure InternalAfterOpen; override;
    procedure InternalClose; override;
    procedure InternalInitFieldDefs; override;
    // edit support
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
    procedure InternalPost; override;
    procedure InternalInsert; override;
    // fields
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    // custom dataset virutal methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord(Buffer: PChar); override;
  public
    procedure CreateTable;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
  published
    property TableName: string read FTableName write SetTableName;
  end;
```

In dividing the methods into sections (as you can see by looking at the source code files), I've marked each one with a roman number. You'll see those numbers in a comment describing the method, so that while browsing this long listing you'll immediately know which of the three sections you are in.

Section I: Initialization, Opening, and Closing

The first methods I'll examine are responsible for initializing the dataset, and for opening and closing the file stream we'll use to store the data. In addition to initializing the component's internal data, these methods are responsible for initializing and connecting the proper TFields objects to the dataset component. To make this work, all we need to do is to initialize the FieldsDef property with the definitions of the fields for our dataset, then call a few standard methods to generate and bind the TField objects. This is the general InternalOpen method:

```
procedure TMDCustomDataSet.InternalOpen;
begin
  InternalPreOpen; // custom method for subclasses

  // initialize the field definitions
  InternalInitFieldDefs;

  // if there are no persistent field objects, create the fields dynamically
  if DefaultFields then
    CreateFields;
  // connect the TField objects with the actual fields
  BindFields (True);

  InternalAfterOpen; // custom method for subclasses

  // sets cracks and record position and size
  BofCrack := -1;
  EofCrack := InternalRecordCount;
  FCurrentRecord := BofCrack;
  FRecordBufferSize := FRecordSize + sizeof (TMdRecInfo);
  BookmarkSize := sizeOf (Integer);

  // everything OK: table is now open
  FIsTableOpen := True;
end;
```

You'll notice that the method sets most of the local fields of the class, and also the BookmarkSize field of the base TDataSet class. Within this method, I call two custom methods I introduced in my custom dataset hierarchy: InternalPreOpen and InternalAfterOpen. The first, InternalPreOpen, is used for operations required at the very beginning, such as checking whether the dataset can actually be opened and reading the header information from the file. The code checks an internal version number for consistency with the value saved when the table is first created, as you'll see later on. By raising an exception in this method, we can eventually stop the open operation.

Here is the code for the to methods in the derived stream-based dataset:

```
const
  HeaderVersion = 10;

procedure TMdDataSetStream.InternalPreOpen;
begin
  // the size of the header
  FDataFileHeaderSize := sizeOf (TMdDataFileHeader);

  // check if the file exists
  if not FileExists (FTableName) then
    raise EMdDataSetError.Create ('Open: Table file not found');

  // create a stream for the file
  FStream := TFileStream.Create (FTableName, fmOpenReadWrite);

  // initialize local data (loading the header)
  FStream.ReadBuffer (FDataFileHeader, FDataFileHeaderSize);
  if FDataFileHeader.VersionNumber <> HeaderVersion then
    raise EMdDataSetError.Create ('Illegal File Version');
  // let's read this, double check later
  FRecordCount := FDataFileHeader.RecordCount;
end;

procedure TMdDataSetStream.InternalAfterOpen;
begin
  // check the record size
  if FDataFileHeader.RecordSize <> FRecordSize then
    raise EMdDataSetError.Create ('File record size mismatch');
  // check the number of records against the file size
 if (FDataFileHeaderSize + FRecordCount * FRecordSize) <> FStream.Size then
    raise EMdDataSetError.Create ('InternalOpen: Invalid Record Size');
end;
```

The second method, `InternalAfterOpen`, is used for operations required after the field definitions have been set and is followed by code that compares the record size read from the file against the value computed in the `InternalInitFieldDefs` method. The code checks also that the number of records read from the header is compatible with the actual size of the file. This test might fail if the dataset wasn't closed properly: you might want to modify this code to let the dataset refresh the record size in the header anyway.

The `InternalOpen` method of the custom dataset class is specifically responsible for calling `InternalInitFieldDefs`, which determines the field definitions (at either design time or run time). For this example, I've decided to base the field definitions on an external file, a simple INI file that provides a section for every field. Each section contains the name and data type of the field, as well as its size if it is string data. Listing 18.3 is the Contrib.INI file that we'll use in the component's demo application:

Listing 18.3: The Contrib.INI file for the demo application

```
[Fields]
Number = 6

[Field1]
Type = ftString
Name = Name
Size = 30

[Field2]
Type = ftInteger
Name = Level

[Field3]
Type = ftDate
Name = BirthDate

[Field4]
Type = ftCurrency
Name = Stipend

[Field5]
Type = ftString
Name = Email
Size = 50

[Field6]
Type = ftBoolean
Name = Editor
```

This file, or a similar one, must use the same name as the table file and must be in the same directory. The InternalInitFieldDefs method (shown in Listing 18.4) will read it, using the values it finds to set up the field definitions and determine the size of each record. The method also initializes an internal TList object that stores the offset of every field inside the record. We'll use this TList to access fields' data within the record buffer, as you can see in the code listing.

Listing 18.4: The InternalInitFieldDefs method of the stream-based dataset

```
procedure TMdDataSetStream.InternalInitFieldDefs;
var
  IniFileName, FieldName: string;
  IniFile: TIniFile;
```

```
    nFields, I, TmpFieldOffset, nSize: Integer;
    FieldType: TFieldType;
  begin
    FFieldOffset := TList.Create;
    FieldDefs.Clear;
    TmpFieldOffset := 0;
    IniFilename := ChangeFileExt(FTableName, '.ini');
    Inifile := TIniFile.Create (IniFilename);
    // protect INI file
    try
      nFields := IniFile.ReadInteger (' Fields', 'Number', 0);
      if nFields = 0 then
        raise EDataSetOneError.Create (' InitFieldsDefs: 0 fields?');
      for I := 1 to nFields do
      begin
        // create the field
        FieldType := TFieldType (GetEnumValue (TypeInfo (TFieldType),
          IniFile.ReadString ('Field' + IntToStr (I), 'Type', '')));
        FieldName := IniFile.ReadString ('Field' + IntToStr (I), 'Name', '');
        if FieldName = ''  then
          raise EDataSetOneError.Create (
            'InitFieldsDefs: No name for field ' + IntToStr (I));
        nSize := IniFile.ReadInteger ('Field' + IntToStr (I), 'Size', 0);
        FieldDefs.Add (FieldName, FieldType, nSize, False);
        // save offset and compute size
        FFieldOffset.Add (Pointer (TmpFieldOffset));
        case FieldType of
          ftString:                     Inc (TmpFieldOffset, nSize + 1);
          ftBoolean, ftSmallInt, ftWord:    Inc (TmpFieldOffset, 2);
          ftInteger, ftDate, ftTime:        Inc (TmpFieldOffset, 4);
          ftFloat, ftCurrency, ftDateTime:  Inc (TmpFieldOffset, 8);
        else
          raise EDataSetOneError.Create (
             'InitFieldsDefs: Unsupported field type');
        end;
      end; // for
    finally
      IniFile.Free;
    end;
    FRecordSize := TmpFieldOffset;
  end;
```

Closing the table is simply a matter of disconnecting the fields (using some standard calls). Each class must dispose the data it allocated and update the file header, the first time records are added and each time the record count has changed:

```
  procedure TMDCustomDataSet.InternalClose;
  begin
    // disconnect field objects
    BindFields (False);
    // destroy field object (if not persistent)
    if DefaultFields then
      DestroyFields;
    // close the file
    FIsTableOpen := False;
  end;

  procedure TMdDataSetStream.InternalClose;
  begin
    // if required, save updated header
    if (FDataFileHeader.RecordCount <> FRecordCount) or
      (FDataFileHeader.RecordSize = 0) then
    begin
      FDataFileHeader.RecordSize := FRecordSize;
      FDataFileHeader.RecordCount := FRecordCount;
      if Assigned (FStream) then
      begin
        FStream.Seek (0, soFromBeginning);
        FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
      end;
    end;
    // free the internal list field offsets and the stream
    FFieldOffset.Free;
    FStream.Free;
    inherited InternalClose;
  end;
```

Another related function is used to test whether the dataset is open, something we can solve using the corresponding local field:

```
  function TMDCustomDataSet.IsCursorOpen: Boolean;
  begin
    Result := FIsTableOpen;
  end;
```

These are the opening and closing methods you need to implement in any custom dataset. However, most of the time, you'll also add a method to create the table. In this example, the `CreateTable` method creates an empty file and inserts information in the header: a fixed version number, a dummy record size (we don't know the size until we initialize the fields), and the record count (which is zero to start):

```
  procedure TMdDataSetStream.CreateTable;
  begin
    CheckInactive;
    InternalInitFieldDefs;

    // create the new file
    if FileExists (FTableName) then
```

```
      raise EMdDataSetError.Create ('File ' + FTableName + ' already exists');
    FStream := TFileStream.Create (FTableName, fmCreate or fmShareExclusive);
    try
      // save the header
      FDataFileHeader.VersionNumber := HeaderVersion;
      FDataFileHeader.RecordSize := 0;    // used later
      FDataFileHeader.RecordCount := 0;   // empty
      FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
    finally
      // close the file
      FStream.Free;
    end;
  end;
```

Section II: Movement and Bookmark Management

As mentioned earlier, one of the things every dataset must implement is *bookmark management*, which is necessary for navigating through the dataset. Logically, a bookmark is a reference to a specific record of the dataset, something that uniquely identifies the record so that a dataset can access it and compare it to other records. Technically, bookmarks are pointers. You can implement them as pointers to specific data structures that store record information, or you can implement them as simple record numbers. For simplicity, I'll use the latter approach.

Given a bookmark, you should be able to find the corresponding record, but given a record buffer, you should also be able to retrieve the corresponding bookmark. This is the reason for appending the TMdRecInfo structure to the record data in each record buffer. This data structure stores the bookmark for the record in the buffer, as well as some bookmark flags defined as:

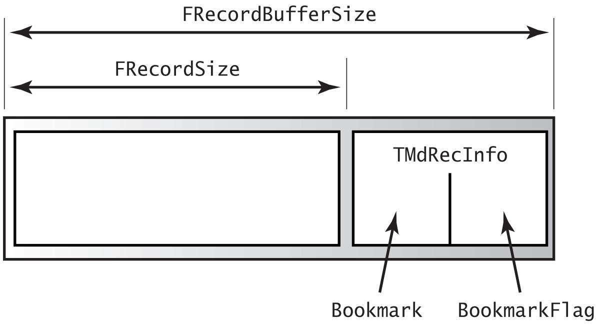
```
type
  TBookmarkFlag = (bfCurrent, bfBOF, bfEOF, bfInserted);
```

The system will request us to store these flags in each record buffer, and will later ask us to retrieve the flags for a given record buffer.

To summarize, the structure of a record buffer stores the data of the record, the bookmark, and the bookmark flags, as you can see in Figure 18.5.

FIGURE 18.5:

The structure of each buffer of the custom dataset, along with the various local fields referring to its sub-portions



To access the bookmark and flags, we can simply use as an offset the size of the actual data, casting the value to the `PMdRecInfo` pointer type, and then access the proper field of the `TMdRecInfo` structure via the pointer. The two methods used to set and get the bookmark flags demonstrate this technique:

```
procedure TMDCustomDataSet.SetBookmarkFlag (Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PMdRecInfo(Buffer + FRecordSize).BookmarkFlag := Value;
end;

function TMDCustomDataSet.GetBookmarkFlag (Buffer: PChar): TBookmarkFlag;
begin
  Result := PMdRecInfo(Buffer + FRecordSize).BookmarkFlag;
end;
```

The methods we use to set and get the current bookmark of a record are similar to the previous two, but they add some complexity because we receive a pointer to the bookmark in the `Data` parameter. Casting this pointer as an integer pointer (`PInteger`) and dereferencing it, we obtain the bookmark value:

```
procedure TMDCustomDataSet.GetBookmarkData (Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ := PMdRecInfo(Buffer + FRecordSize).Bookmark;
end;

procedure TMDCustomDataSet.SetBookmarkData (Buffer: PChar; Data: Pointer);
begin
  PMdRecInfo(Buffer + FRecordSize).Bookmark := PInteger(Data)^;
end;
```

The key bookmark management method is `InternalGotoBookmark`, which your dataset uses to make a given record the current one. You'll notice that this isn't the standard navigation technique, since it's much more common to move to the next or previous record (something we can accomplish using the GetRecord method presented in the next section), or to move to the first or last record (something we'll accomplish using the InternalFirst and InternalLast methods described shortly).

Oddly enough, the InternalGotoBookmark method doesn't expect a bookmark parameter, but a pointer to a bookmark, so we must dereference it to determine the bookmark value. The following method, InternalSetToRecord, is what you use to jump to a given bookmark, but it must extract the bookmark from the record buffer passed as a parameter. Then, InternalSetToRecord calls InternalGotoBookmark. Here are the two methods:

```
procedure TMDCustomDataSet.InternalGotoBookmark (Bookmark: Pointer);
var
  ReqBookmark: Integer;
begin
  ReqBookmark := PInteger (Bookmark)^;
  if (ReqBookmark >= 0) and (ReqBookmark < InternalRecordCount) then
    FCurrentRecord := ReqBookmark
  else
    raise EMdDataSetError.Create ('Bookmark ' +
      IntToStr (ReqBookmark) + ' not found');
end;

procedure TMDCustomDataSet.InternalSetToRecord (Buffer: PChar);
var
  ReqBookmark: Integer;
begin
  ReqBookmark := PMdRecInfo(Buffer + FRecordSize).Bookmark;
  InternalGotoBookmark (@ReqBookmark);
end;
```

In addition to the bookmark management methods just described, there are several other navigation methods we use to move to specific positions within the dataset, such as the first or last record. Actually, these two methods don't move the current record pointer to the first or last record, but move it to one of two special locations before the first record and after the last one. These are not actual records; Borland calls them *cracks*. The beginning-of-file crack, or BofCrack, has the value –1 (set in the InternalOpen method), since the position of the first record is zero. The end-of-file crack, or EofCrack, has the value of the number of records, since the last record has the position FRecordCount - 1. We've used two local fields, called EofCrack and BofCrack, to make this code easier to read:

```
procedure TMDCustomDataSet.InternalFirst;
begin
  FCurrentRecord := BofCrack;
end;

procedure TMDCustomDataSet.InternalLast;
```

```
begin
  EofCrack := InternalRecordCount;
  FCurrentRecord := EofCrack;
end;
```

The `InternalRecordCount` method is a virtual method introduced in my `TMDCustomDataSet` class, as different datasets can either have a local field for this value (as in case of the stream-based dataset, which has an `FRecordCount` field) or compute it on-the-fly.

Another group of optional methods is used to get the current record number (used by the DBGrid component to show a proportional vertical scroll bar), set the current record number, or determine the number of records. These methods are quite easy to understand, if you recall that the range of the internal `FCurrentRecord` field is from 0 to the number of records minus 1. In contrast, the record number reported to the system ranges from 1 to the number of records:

```
function TMDCustomDataSet.GetRecordCount: Longint;
begin
  CheckActive;
  Result := InternalRecordCount;
end;

function TMDCustomDataSet.GetRecNo: Longint;
begin
  UpdateCursorPos;
  if FCurrentRecord < 0 then
    Result := 1
  else
    Result := FCurrentRecord + 1;
end;

procedure TMDCustomDataSet.SetRecNo(Value: Integer);
begin
  CheckBrowseMode;
  if (Value > 1) and (Value <= FRecordCount) then
  begin
    FCurrentRecord := Value - 1;
    Resync([]);
  end;
end;
```

Notice that it is the generic custom dataset class that implements all the methods of this section. The derived stream-based dataset doesn't need to modify any of them.

Section III: Record Buffers and Field Management

Now that we've covered all the support methods, we can examine the core of a custom dataset. Besides opening and creating records and moving around between them, the component really needs to move the data from the stream (the persistent file) to the record buffers, and from the record buffers to the `TField` objects that are connected to the data-aware controls. The management of record buffers is quite complex, because each dataset also needs to allocate, empty, and free the memory it requires:

```
function TMDCustomDataSet.AllocRecordBuffer: PChar;
begin
  GetMem (Result, FRecordBufferSize);
end;

procedure TMDCustomDataSet.FreeRecordBuffer (var Buffer: PChar);
begin
  FreeMem (Buffer);
end;
```

The reason for allocating memory this way is that a dataset generally adds more information to the record buffer, so the system has no way of knowing how much memory to allocate. You'll notice that in the `AllocRecordBuffer` method, the component allocates the memory for the record buffer, including both the database data and the record information. In fact, in the `InternalOpen` method I wrote

```
FRecordBufferSize := InternalRecordSize + sizeof (TMdRecInfo);
```

The component also needs to implement a function to reset the buffer, `InternalInitRecord`, usually filling it with numeric zeros or spaces.

Oddly enough, we must also implement a method that returns the size of each record, but only the data portion—not the entire record buffer. This method is necessary for implementing the read-only `RecordSize` property, used only in a couple of peculiar cases in the entire VCL source code. In the generic custom dataset, the `GetRecordSize` method returns the value of the `FRecordSize` field.

Now we've actually reached the core of our custom dataset component. The methods of this group are `GetRecord`, which reads data from the file, `InternalPost` and `InternalAddRecord`, which update or add new data to the file, and `InternalDelete`, which removes data and is not implemented in my sample dataset.

The most complex method of this group is probably `GetRecord`, which serves multiple purposes. In fact, this method is used by the system to retrieve the data for the current record, fill a buffer passed as a parameter, and retrieve the data of the next or previous records. The `GetMode` parameter determines its action:

```
type
  TGetMode = (gmCurrent, gmNext, gmPrior);
```

Of course, a previous or next record might not exist. Even the current record might not exist; for example, when the table is empty (or in case of an internal error). In these cases we don't retrieve the data but return an error code. Therefore, this method's result can be one of the following values:

```
type
  TGetResult = (grOK, grBOF, grEOF, grError);
```

Checking to see if the requested record exists is slightly different than you might expect. We don't have to determine if the current record is in the proper range, only if the requested one is. For example, in the gmCurrent branch of the case statement, we use the standard expression CurrentRecord>=InternalRecourdCount. To fully understand the various cases, you might want to read the code a couple of times.

It took me some trial and error (and system crashes caused by recursive calls) to get it straight when I wrote my first custom dataset a few years back. To test it, consider that if you use a DBGrid, the system will perform a series of GetRecord calls, until either the grid is full or GetRecord return grEOF. Here's the entire code of the GetRecord method:

```
// III: Retrieve data for current, previous, or next record
// (moving to it if necessary) and return the status
function TMdCustomDataSet.GetRecord(Buffer: PChar;
  GetMode: TGetMode; DoCheck: Boolean): TGetResult;
begin
  Result := grOK; // default
  case GetMode of
    gmNext: // move on
      if FCurrentRecord < InternalRecordCount - 1 then
        Inc (FCurrentRecord)
      else
        Result := grEOF; // end of file
    gmPrior: // move back
      if FCurrentRecord > 0 then
        Dec (FCurrentRecord)
      else
        Result := grBOF; // begin of file
    gmCurrent: // check if empty
      if FCurrentRecord >= InternalRecordCount then
        Result := grError;
  end;
  // load the data
  if Result = grOK then
    InternalLoadCurrentRecord (Buffer)
  else if (Result = grError) and DoCheck then
    raise EMdDataSetError.Create ('GetRecord: Invalid record');
end;
```

If there's an error and the DoCheck parameter was True, GetRecord raises an exception. If everything goes fine during record selection, the component loads the data from the stream, moving to the position of the current record (given by the record size multiplied by the record number). In addition, we need to initialize the buffer with the proper bookmark flag and bookmark (or record number) value. This is accomplished by another virtual method I introduced, so that derived classes will only need to implement this portion of the code, while the complex GetRecord method remains unchanged:

```
procedure TMdDataSetStream.InternalLoadCurrentRecord (Buffer: PChar);
begin
  FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
  FStream.ReadBuffer (Buffer^, FRecordSize);
  with PMdRecInfo(Buffer + FRecordSize)^ do
  begin
    BookmarkFlag := bfCurrent;
    Bookmark := FCurrentRecord;
  end;
end;
```

We move data to the file in two different cases: when you modify the current record (that is, a post after an edit) or when you add a new record (a post after an insert or append). We use the InternalPost method in both cases, but we can check the dataset's State property to determine which type of post we're performing. In both cases we don't receive a record buffer as a parameter, so we must use the ActiveRecord property of TDataSet, which points to the buffer for the current record:

```
procedure TMdDataSetStream.InternalPost;
begin
  CheckActive;
  if State = dsEdit then
  begin
    // replace data with new data
    FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
  end
  else
  begin
    // always append
    InternalLast;
    FStream.Seek (0, soFromEnd);
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
    Inc (FRecordCount);
  end;
end;
```

In addition, there's another related method, InternalAddRecord. This method is called by the AddRecord method, which in turn is called by InsertRecord and AppendRecord. These last two are public methods a user can call. This is an alternative to inserting or appending a new record to the dataset, editing the values of the various fields, and then posting the data, since the InsertRecord and AppendRecord calls receive the values of the fields as parameters. All we must do at that point is replicate the code used to add a new record in the InternalPost method:

```
procedure TMdDataSetOne.InternalAddRecord(Buffer: Pointer; Append: Boolean);
begin
  // always append at the end
  InternalLast;
  FStream.Seek (0, soFromEnd);
  FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
  Inc (FRecordCount);
end;
```

The last file operation I should have implemented is one that removes the current record. This operation is common, but it is actually quite complex. If we take a simple approach, such as creating an empty spot in the file, then we'll need to keep track of that spot and make the code for reading or writing a specific record work around that spot. An alternate solution is to make a copy of the entire file, without the given record, and then replace the original file with the copy. Given these choices, I felt that for this example I could forgo supporting record deletion.

Section IV: From Buffers to Fields

In the last few methods, we've seen how datasets move data from the data file to the memory buffer. However, there's little Delphi can do with this record buffer, because it doesn't yet know how to interpret the data in the buffer. We need to provide two more methods: GetData, which copies the data from the record buffer to the field objects of the dataset, and SetData, which moves the data back from the fields to the record buffer. What Delphi will do automatically for us is move the data from the field objects to the data-aware controls, and back.

The code for these two methods isn't very complex, primarily because we saved the field offsets inside the record data in a TList object called FFieldOffset. By simply incrementing the pointer to the initial position in the record buffer of the current field's offset, we'll be able to get the specific data, which takes Field.DataSize bytes.

A confusing element of these two methods is that they both accept a Field parameter and a Buffer parameter. At first, one might think that the buffer passed as parameter is the record buffer. Actually, I found out that the Buffer is a pointer to the field object's raw data. If you use one of the field object's methods to move that data, it will call the dataset's GetData or

SetData methods, probably causing an infinite recursion. Instead, you should use the ActiveBuffer pointer to access the record buffer, use the proper offset to get to the data for the current field in the record buffer, and then use the provided Buffer to access the field data. The only difference between the two methods is the direction we're moving the data:

```
function TMdDataSetOne.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FieldOffset: Integer;
  Ptr: PChar;
begin
  Result := False;
  if not IsEmpty and (Field.FieldNo > 0) then
  begin
    FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
    Ptr := ActiveBuffer;
    Inc (Ptr, FieldOffset);
    if Assigned (Buffer) then
      Move (Ptr^, Buffer^, Field.DataSize);
    Result := True;
    if (Field is TDateTimeField) and (PInteger(Ptr)^ = 0) then
      Result := False;
  end;
end;

procedure TMdDataSetOne.SetFieldData(Field: TField; Buffer: Pointer);
var
  FieldOffset: Integer;
  Ptr: PChar;
begin
  if Field.FieldNo >= 0 then
  begin
    FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
    Ptr := ActiveBuffer;
    Inc (Ptr, FieldOffset);
    if Assigned (Buffer) then
      Move (Buffer^, Ptr^, Field.DataSize)
    else
      raise Exception.Create (
        'Very bad error in TMdDataSetStream.SetField data');
    DataEvent (deFieldChange, Longint(Field));
  end;
end;
```

The GetField method should return True or False to indicate whether the field contains data or is empty. However, unless you use a special marker for blank fields, it's very difficult to determine this, since we're storing values of different data types. For example, a test such as `Ptr^<>#0` makes sense only if you are using a string representation for all of the fields. If you use this test, zero integer values and empty strings will show as null values (the data-aware controls will be empty), which may be what you want. The problem is that Boolean False values won't show up. Even worse, floating-point values with no decimals and few digits won't be displayed, because the exponent portion of their representation will be zero! However, to make this example work in Delphi 6, I had to consider as empty each date/time field with an initial zero. Without this code, Delphi tries to convert the illegal *internal* zero date (internally, date fields don't use a `TDateTime` data type but a different representation) raising an exception. The code used to work with past versions of Delphi.

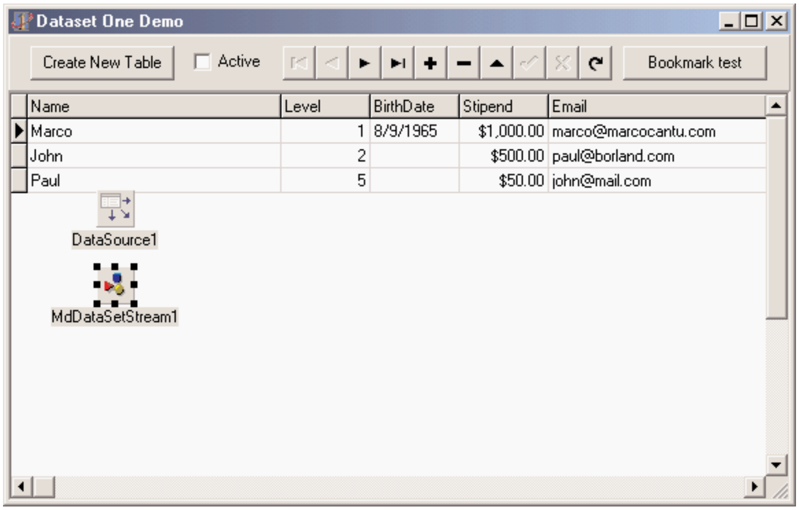
WARNING While trying to fix this problem, I also found out that if you call `IsNull` for a field, this request is resolved by calling `GetFieldData` without passing any buffer to fill but looking only for the result of the function call. This is the reason for the `if Assigned (Buffer)` test within the code.

There's one final method, which doesn't fall into any category: `InternalHandleException`. Generally, this method silences the exception, as it is activated only at design time.

Testing the Stream-Based DataSet

After all this work, we're finally ready to test an application example of the custom dataset component, installed in the component's package for this chapter. The form displayed by the StreamDSDemo program is quite simple, as you can see in Figure 18.6. It has a panel with two buttons, a check box, and a navigator component, plus a DBGrid filling its client area.

FIGURE 18.6: The form of the StreamDS-Demo example. The custom dataset has been activated, so we can already see the data at design time.



Name	Level	BirthDate	Stipend	Email
Marco	1	8/9/1965	$1,000.00	marco@marcocantu.com
John	2		$500.00	paul@borland.com
Paul	5		$50.00	john@mail.com

Figure 18.6 shows the form of the example at design time, but we've activated the custom dataset so that its data is already visible. Of course we'd already prepared the INI file with the table definition (it's the file we already listed when discussing the dataset initialization), and we executed the program to add some data to the file.

It's also possible to modify the form using Delphi's Fields editor and set the properties of the various field objects. Everything works as it does with one of the standard dataset controls! However, to make this work you'll need to enter the name of the custom dataset's file in the `TableName` property, using the complete path.

WARNING As the demo program defines the absolute path of the table file at design time, you'll need to fix it if you copy the examples to a different drive or directory. In the example, the `TableName` property is used only at design time. At run time, in fact, the program looks for the table in the current directory.

The code of the example is rather simple, especially compared to the code of the custom dataset. If the table doesn't exist yet, you can click the Create button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MdDataSetStream1.CreateTable;
  MdDataSetStream1.Open;
  CheckBox1.Checked := MdDataSetStream1.Active;
end;
```

You'll notice that we create the file first, open and close it, and then open the table. This is the same behavior as the `TTable` component (which accomplishes this using the `CreateTable` method). To simply open or close the table, you can click the check box:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  MdDataSetStream1.Active := CheckBox1.Checked;
end;
```

Finally, I've created a method that tests custom dataset's bookmark management code and seems to work.

A Directory in a Dataset

An important idea related to datasets in Delphi is that they simply represent a set of data, regardless where this data comes from. An SQL server or a local file are examples of traditional datasets, but you can use the same technology to show a list of users of a system, a list of files of a folder, the properties of some objects, some XML-based data, and so on.

As an example, the second (and last) dataset presented in this chapter is a list of files. This is based once more on a generic approach. I've built a generic dataset based on a list of objects in memory (using a TObjectList), then derived a version in which the objects correspond to the files of a folder. The actual example is simplified by the fact it is a read-only dataset, so you might even find it simpler than the previous dataset I presented.

NOTE Some of the ideas presented here were discussed in an article I wrote for the Borland Community Web site, http://community.borland.com, published in June 2000.

A List as a Dataset

The generic list-based dataset is called TMdListDataSet and contains the list of objects, created when you open the dataset and freed when you close it. This dataset doesn't store the actual record data within the buffer; rather, it saves in the buffer only the position in the list of the entry corresponding to the record's data. This is the class definition:

```
type
  TMdListDataSet = class (TMdCustomDataSet)
  protected
    // the list holding the data
    FList: TObjectList;
    // dataset virtual methods
    procedure InternalPreOpen; override;
    procedure InternalClose; override;
    // custom dataset virtual methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord (Buffer: PChar); override;
  end;
```

You can see that by writing a generic custom data class, we can override few virtual methods of the TDataSet class and of this custom dataset class, and have a working dataset (although this is still an abstract class, which requires extra code from subclasses to work). When the dataset is opened, we have to create the list and set the record size, to indicate we're simply saving the list index in the buffer:

```
procedure TMdListDataSet.InternalPreOpen;
begin
  FList := TObjectList.Create (True); // owns the objects
  FRecordSize := 4; // an integer, the list item id
end;
```

Further subclasses at this point should also fill the list with actual objects.

TIP

Similarly to the ClientDataSet, my list dataset keeps all of its data in memory. However, using some smart techniques, you can also create a list of “fake” objects, and then load the actual objects only when you are accessing them.

Closing is simply a matter of freeing the list, which has a record count corresponding to the list size:

```
function TMdListDataSet.InternalRecordCount: Integer;
begin
  Result := fList.Count;
end;
```

The only other method is used to save the data of the current record in the record buffer, including the bookmark information. The core data is simply the position of the current record, which matches the list index (and also the bookmark):

```
procedure TMdListDataSet.InternalLoadCurrentRecord (Buffer: PChar);
begin
  PInteger (Buffer)^ := fCurrentRecord;
  with PMdRecInfo(Buffer + FRecordSize)^ do
  begin
    BookmarkFlag := bfCurrent;
    Bookmark := fCurrentRecord;
  end;
end;
```

Directory Data

The derived directory dataset class has to provide a way to load the objects in memory when the dataset is opened, to define the proper fields, and to read and write the value of those fields. Of course, it has also a property indicating the directory to work on, or to be more precise, the directory plus the file mask used for filtering the files (as in `c:\docs\*.txt`):

```
type
  TMdDirDataset = class(TMdListDataSet)
  private
    FDirectory: string;
    procedure SetDirectory(const NewDirectory: string);
  protected
    // TDataSet virtual methdos
    procedure InternalInitFieldDefs; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    function GetCanModify: Boolean; override;
    // custom dataset virtual methods
    procedure InternalAfterOpen; override;
  public
```

```
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
  published
    property Directory: string read FDirectory write SetDirectory;
  end;
```

The GetCanModify function is another virtual method of TDataSet, used to determine if the dataset is read-only. In this case, it simply returns False. Also, we won't have to write any code for the SetFieldData procedure, but we have to define it because it is an abstract virtual method.

As I am dealing with a list of objects, the unit includes also a class for those objects. In this case, I am working with file data extracted by a TSearchRec buffer by the TFileData class constructor:

```
type
  TFileData = class
  public
    ShortFileName: string;
    Time: TDateTime;
    Size: Integer;
    Attr: Integer;
    constructor Create (var FileInfo: TSearchRec);
  end;

constructor TFileData.Create (var FileInfo: TSearchRec);
begin
  ShortFileName := FileInfo.Name;
  Time := FileDateToDateTime (FileInfo.Time);
  Size := FileInfo.Size;
  Attr := FileInfo.Attr;
end;
```

This constructor is called for each folder while opening the dataset:

```
procedure TMdDirDataset.InternalAfterOpen;
var
  Attr: Integer;
  FileInfo: TSearchRec;
  FileData: TFileData;
begin
  // scan all files
  Attr := faAnyFile;
  FList.Clear;
  if SysUtils.FindFirst(fDirectory, Attr, FileInfo) = 0 then
  repeat
    FileData := TFileData.Create (FileInfo);
    FList.Add (FileData);
  until SysUtils.FindNext(FileInfo) <> 0;
  SysUtils.FindClose(FileInfo);
end;
```

The next step is to define the fields of the dataset, which in this case are fixed and depend on the available directory data:

```
procedure TMdDirDataset.InternalInitFieldDefs;
begin
  if fDirectory = '' then
    raise EMdDataSetError.Create ('Missing directory');

  // field definitions
  FieldDefs.Clear;
  FieldDefs.Add ('FileName', ftString, 40, True);
  FieldDefs.Add ('TimeStamp', ftDateTime);
  FieldDefs.Add ('Size', ftInteger);
  FieldDefs.Add ('Attributes', ftString, 3);
  FieldDefs.Add ('Folder', ftBoolean);
end;
```

Finally, the component has to move the data from the object of the list referenced by the current record buffer (the `ActiveBuffer` value) to each field of the dataset, as requested by the `GetFieldData` method. This function uses either `Move` or `StrCopy`, depending on the data type and does some conversions for the attributes codes (*H* for hidden, *R* for read-only, and *S* for system) extracted from the related flags and used also to determine whether a file is actually a folder. Here is the code:

```
function TMdDirDataset.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FileData: TFileData;
  Bool1: WordBool;
  strAttr: string;
  t: TDateTimeRec;
begin
  FileData := fList [PInteger(ActiveBuffer)^] as TFileData;
  case Field.Index of
    0: // filename
      StrCopy (Buffer, pchar(FileData.ShortFileName));
    1: // timestamp
    begin
      t := DateTimeToNative (ftdatetime, FileData.Time);
      Move (t, Buffer^, sizeof (TDateTime));
    end;
    2: // size
      Move (FileData.Size, Buffer^, sizeof (Integer));
    3: // attributes
    begin
      strAttr := '   ';
      if (FileData.Attr and SysUtils.faReadOnly) > 0 then
        strAttr [1] := 'R';
        if (FileData.Attr and SysUtils.faSysFile) > 0 then
          strAttr [2] := 'S';
        if (FileData.Attr and SysUtils.faHidden) > 0 then
          strAttr [3] := 'H';
        StrCopy (Buffer, pchar(strAttr));
      end;
      4: // folder
      begin
        Bool1 := FileData.Attr and SysUtils.faDirectory > 0;
        Move (Bool1, Buffer^, sizeof (WordBool));
      end;
    end; // case
    Result := True;
  end;
```

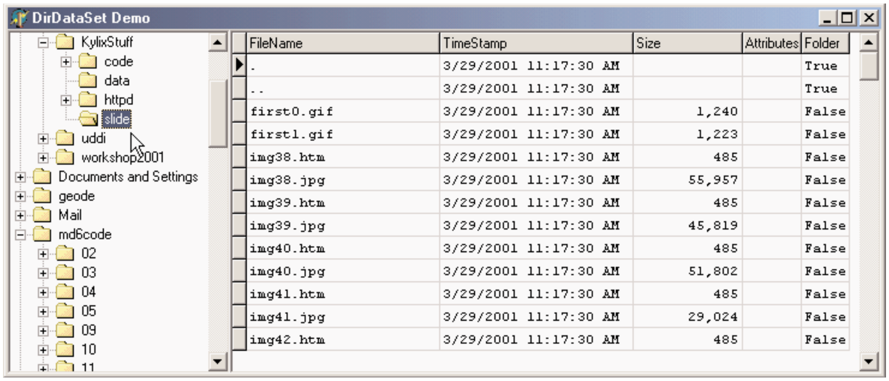
The tricky part in writing this code was figuring out the internal format of dates stored within date/time fields. This is not the common TDateTime format used by Delphi, and not even the internal TTimeStamp, but what is called the internally called the "native" date time format. I've written a conversion function cloning one I've found in the VCL code for the date/time fields:

```
function DateTimeToNative(DataType: TFieldType; Data: TDateTime): TDateTimeRec;
var
  TimeStamp: TTimeStamp;
begin
  TimeStamp := DateTimeToTimeStamp(Data);
  case DataType of
    ftDate: Result.Date := TimeStamp.Date;
    ftTime: Result.Time := TimeStamp.Time;
  else
    Result.DateTime := TimeStampToMSecs(TimeStamp);
  end;
end;
```

With this dataset available, building the demo program (shown in Figure 18.7) was simply a matter of connecting a DBGrid component to it and adding a folder-selection component, Delphi 6's ShellTreeView control. This control is set up to work only on files, by setting its Root property to C:\. When the user selects a new folder, the OnChange event handler of the ShellTreeView control refreshes the dataset.

FIGURE 18.7:

The output of the DirDemo example, which uses a rather unusual dataset, showing directory data.



FileName	TimeStamp	Size	Attributes	Folder
.	3/29/2001 11:17:30 AM			True
..	3/29/2001 11:17:30 AM			True
first0.gif	3/29/2001 11:17:30 AM	1,240		False
first1.gif	3/29/2001 11:17:30 AM	1,223		False
img38.htm	3/29/2001 11:17:30 AM	485		False
img38.jpg	3/29/2001 11:17:30 AM	55,957		False
img39.htm	3/29/2001 11:17:30 AM	485		False
img39.jpg	3/29/2001 11:17:30 AM	45,819		False
img40.htm	3/29/2001 11:17:30 AM	485		False
img40.jpg	3/29/2001 11:17:30 AM	51,802		False
img41.htm	3/29/2001 11:17:30 AM	485		False
img41.jpg	3/29/2001 11:17:30 AM	29,024		False
img42.htm	3/29/2001 11:17:30 AM	485		False

```
procedure TForm1.ShellTreeView1Change(Sender: TObject; Node: TTreeNode);
begin
  MdDirDataset1.Close;
  MdDirDataset1.Directory := ShellTreeView1.Path + '\*.*';
  MdDirDataset1.Open;
end;
```

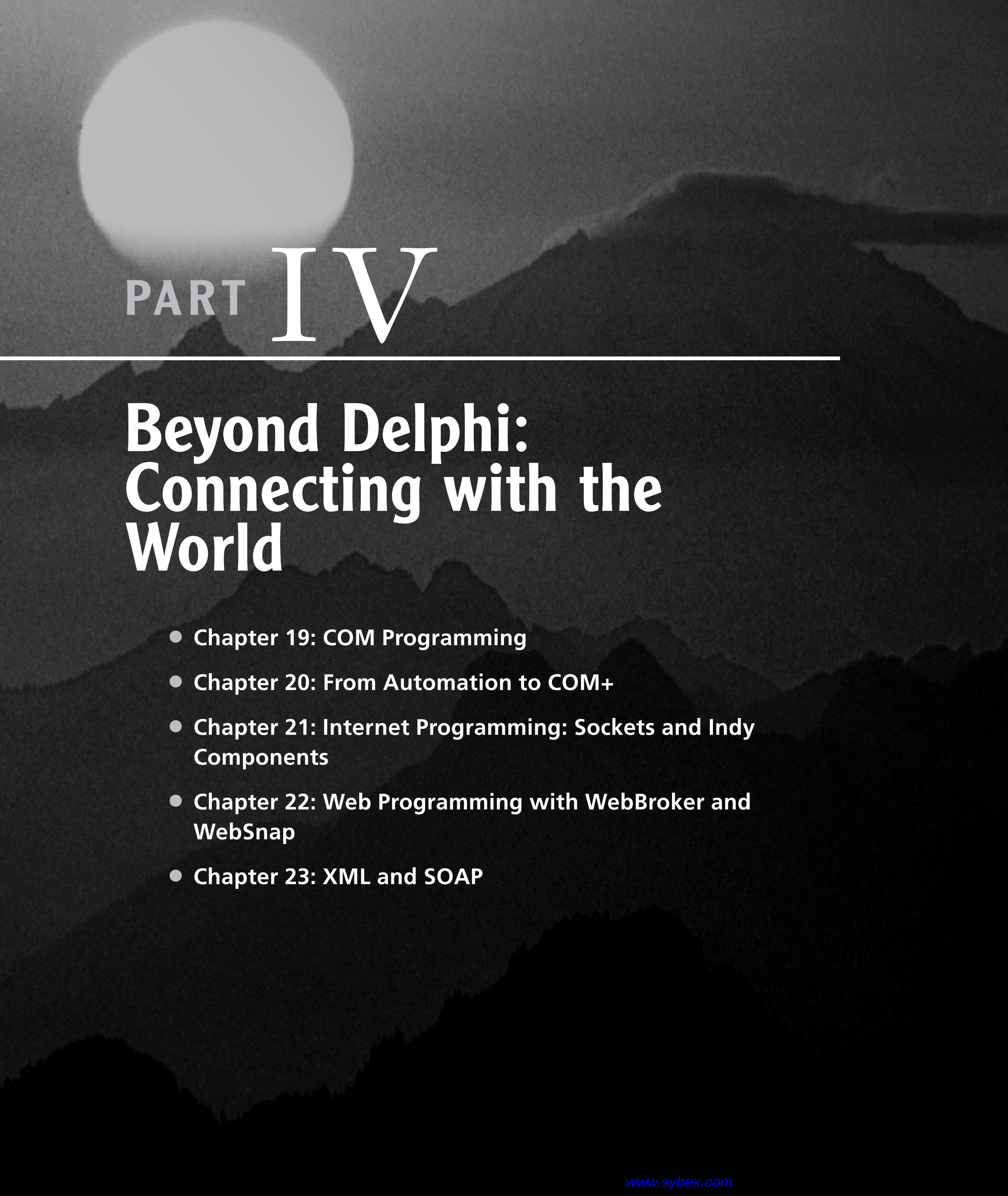
What’s Next?

In this chapter we’ve delved inside Delphi’s database architecture, by first examining the development of data-aware controls and then studying the internals of the `TDataSet` class to write a couple of custom dataset components. With this information, and all the other ideas presented in this part devoted to database programming, you should probably be able to choose the architecture of your database applications, depending on your needs.

NOTE I’ve actually extended the list-based dataset to build an object-based version, hosting the business logic of an application and mapped to a relational database. Refer to my Web site (`www.marcocantu.com`) or contact me for the availability of this code.

Database programming is certainly a core element of Delphi, the reason for devoting several chapters of the book to this topic. We’ll get back to this topic when focusing on presenting database data over the Web, in Chapters 21 and 22.

For the moment, though, we have to introduce another important element of Windows applications, COM and OLE Automation, covered in the next two chapters.



PART IV

Beyond Delphi: Connecting with the World

CHAPTER 19

COM Programming

- What are OLE and COM?
- COM, GUIDs, and class factories
- Delphi interfaces and COM
- The VCL COM-support classes
- Windows shell interfaces

For about 10 years, starting soon after the release of Windows 3.0, Microsoft has kept promising that its operating system and their API would be based on a real object model instead of functions. According to the speculations, Windows 95 (and later Windows 2000) should have been based on this revolutionary approach. Nothing like this happened, but Microsoft kept pushing COM (Component Object Model), built the Windows 95 shell on top of it, pushed applications integration with COM and derivative technologies (such as Automation), and reached the peak by introducing COM+ with Windows 2000.

Now, soon after the release of the complete foundation required for high-level COM programming, Microsoft has decided to switch to a new core technology, part of the dotNet (or .Net, if you prefer) initiative. My impression is that COM wasn't really suited for the integration of fine-grained objects, though it succeeded in providing an architecture for integrating applications or large objects.

NOTE dotNet is a mix of interesting new technologies and pure marketing hype, and this book doesn't discuss it in detail. Even if it were possible to predict how dotNet will affect programmers using Microsoft development tools, it is far from clear how it will affect Delphi programmers. dotNet, in fact, is based on a class library very similar to Delphi's VCL, and it is unclear what the advantage will be of switching to it. If Borland could bundle the VCL with an operating system, along with its core run-time packages, you'd have a situation very similar to dotNet, as far as the library is concerned. Instead, if you are looking for a virtual machine and portable code, you can certainly consider Java and the portability between Windows and Linux possible with the CLX library of Delphi 6 and Kylix. Having said this, I'm not underestimating dotNet, but this is a book on Delphi programming. In any case, we'll get back to many core elements of dotNet in the final chapters, discussing XML and SOAP. Finally, the system can expose dotNet objects as COM objects, so after you learn COM you'll also have a chance to interact with dotNet.

In this chapter, we'll build our first COM object and integrate COM objects with the Windows shell. Type libraries, Automation, and other topics will be covered in the next chapter. I will stick to the basic elements to let you understand the role of this technology without delving heavily into the details. I'll bear in mind the clouds on the future of COM, declared obsolete by Microsoft after the announcement of dotNet but still heavily used by the same company inside their applications and operating systems.

A Short History of OLE and COM

Part of the confusion related to COM technology comes from the fact that Microsoft has used different names for it for marketing reasons. Everything started with Object Linking and Embedding (OLE, for short), which was an extension of the DDE (Dynamic Data Exchange) model. Using the Clipboard allows you to copy some raw data, and using DDE allows you to connect parts of two documents. OLE allows you to copy data from a server application to a client application, along with information regarding the server or a reference to information stored in the Windows Registry. The raw data might be copied along with the link (object embedding) or kept in the original file (object linking). OLE Documents are now called *Active Documents*.

Microsoft updated OLE to OLE 2 and started adding new features, such as OLE Automation and OLE Controls. The next step was to build the Windows 95 shell using OLE technology and interfaces and then to rename the OLE Controls (previously known also as OCX) as ActiveX controls, changing the specification to allow for lightweight controls suitable for distribution over the Internet. For a while, Microsoft promoted ActiveX controls as suitable for the Internet, but the idea was never fully accepted by the development community, certainly not as "suitable" for Internet development.

As this technology was extended and became increasingly important to the Windows platform, Microsoft changed the name to OLE, and then to COM, and now to COM+ for Windows 2000. These changes in naming are only partially related to technological changes and are driven to a large extent by marketing purposes.

What, then, is COM? Basically, the Component Object Model, or COM, is a technology that defines a standard way for a client module and a server module to communicate through a specific interface. Here, "module" indicates an application or a library (a DLL); the two modules may execute on the same computer or on different machines connected via a network. Many interfaces are possible, depending on the role of the client and server, and you can add new interfaces for specific purposes. These interfaces are implemented by server objects. A server object usually implements more than one interface, and all the server objects have a few common capabilities, because they must all implement the `IUnknown` interface.

The good news is that Delphi is fully compliant with COM. When you look at the source code, Object Pascal seems to be easier to use than C++ or other languages for writing COM objects. This simplicity mainly derives from the incorporation of interface types into the Object Pascal language. By the way, interfaces are also similarly used to integrate Java with COM on the Windows platform.

The purpose of COM interfaces is to communicate between two software modules, two executable files, or one executable file and a DLL. Implementing COM objects in DLLs is generally simpler, because in Win32, a program and the DLL it uses reside in the same memory address space. This means that if the program passes a memory address to the DLL, the address remains valid. When you use two executable files, COM has a lot of work to do behind the scenes to let the two applications communicate. This mechanism is called *marshaling*. Note that a DLL implementing COM objects is described as an *in-process* server, whereas when the server is a separate executable, it is called an *out-of-process* server. However, when DLLs are executing on another machine (DCOM) or inside a host environment (MTS), they are also out-of-process.

Implementing *IUnknown*

Before we start looking to an example of COM development, I would like to introduce a few COM basics. The first is that every COM object must implement the IUnknown interface, also dubbed IInterface in Delphi 6 for non-COM usage of interfaces (as we saw in Chapter 3, "The Object Pascal Language: Inheritance and Polymorphism"). This is the base interface from which every Delphi interface inherits, and Delphi provides a couple of different classes with ready-to-use implementations of IUnknown/IInterface, including TInterfacedObject and TComObject. The first can be used to have an internal object unrelated with COM, while the second is used to create objects that can be exported by servers. As I'll discuss in Chapter 20, "From Automation to COM+," several other classes inherit from TComObject and provide support for more interfaces, which are required by Automation servers or ActiveX controls.

As mentioned in Chapter 3, the IUnknown interface has three methods: _AddRef, _Release, and QueryInterface. Here is the definition of the IUnknown interface (extracted from the System unit):

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

The _AddRef and _Release methods are used to implement reference counting. The QueryInterface method handles the type information and type compatibility of the objects.

NOTE In the code above, you can see an example of an **out** parameter, a parameter passed back from the method to the calling program but without an initial value passed by the calling program to the method. The **out** parameters have been added to Delphi's Object Pascal language specifically to support COM. It's also important to note that although Delphi's language definition for the interface type is designed for compatibility with COM, Delphi interfaces do not require COM. This was already highlighted in Chapter 3, where I built a complex interface-based example with no COM support whatsoever.

You don't usually need to implement these methods, as you can inherit from one of the Delphi classes already supporting them. The most important class is `TComObject`, defined in the ComObj unit. When you build a COM server, you'll generally inherit from this class. Because `TComObject` is a complex class, this excerpt shows only its key elements:

```
type
  TComObject = class(TObject, IUnknown, ISupportErrorInfo)
  private
    FNonCountedObject: Boolean;
    FRefCount: Integer;
  protected
    { IUnknown }
    function IUnknown.QueryInterface = ObjQueryInterface;
    function IUnknown._AddRef = ObjAddRef;
    function IUnknown._Release = ObjRelease;
    { ISupportErrorInfo }
    function InterfaceSupportsErrorInfo(const iid: TIID): HResult; stdcall;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Initialize; virtual;
    function ObjAddRef: Integer; virtual; stdcall;
    function ObjQueryInterface(const IID: TGUID; out Obj): HResult;
      virtual; stdcall;
    function ObjRelease: Integer; virtual; stdcall;
    property RefCount: Integer read FRefCount;
  end;
```

This class implements the `IUnknown` interface (using the `ObjAddRef`, `ObjQueryInterface`, and `ObjRelease` methods, as indicated by the method-mapping statements in the protected portion of the class) and the `ISupportErrorInfo` interface (through the `InterfaceSupportsErrorInfo` method). The implementation of reference counting for the `TComObject` class has been extended to support threading. Instead of using `Inc` and `Dec`, the code uses the thread-safe `InterlockedIncrement` and `InterlockedDecrement` API functions, as you can see in the source code of the class:

```
function TComObject.ObjAddRef: Integer;
begin
  Result := InterlockedIncrement(FRefCount);
end;

function TComObject.ObjRelease: Integer;
begin
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then Destroy;
end;
```

As you can see, the implementation of Release destroys the object when there are no more references to it. At first sight, the need to call this method each time you operate on an object seems like a lot of work. However, you might remember from Chapter 3 that when you're using interface variables to refer to objects, Delphi automatically adds the reference-counting calls to the compiled code, which automatically destroys unreferenced objects. This labor-saving feature makes Delphi a convenient tool for COM development.

The most complex method is QueryInterface, which in Delphi is actually implemented through the GetInterface method of the TObject class:

```
function TComObject.ObjQueryInterface(const IID: TGUID; out Obj): HResult;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NOINTERFACE;
end;
```

The role of the QueryInterface method is twofold:

- QueryInterface is used for type checking. The program can ask an object the following questions: Are you of the type I'm interested in? Do you implement the interface I want to call? And the specific methods? If the answer is no, the program can look for another object, maybe asking another server.
- If the answer is yes, QueryInterface usually returns a pointer to the object, using its reference output parameter (out).

To understand the role of the QueryInterface method, it is important to keep in mind that a COM object can implement multiple interfaces, as the event TComObject does. When you call QueryInterface, you might ask for one of the possible interfaces of the object, using the TGUID parameter.

Globally Unique Identifiers

The QueryInterface method has a special parameter of the TGUID type. This is an ID that identifies any COM server class and any interface in the system. When you want to know whether an object supports a specific interface, you ask the object whether it implements the interface that has a given ID (which for the default OLE interfaces is determined by Microsoft).

Another ID is used to indicate a specific class, a specific server. The Windows Registry stores this ID, with indications of the related DLL or executable file. The developers of an OLE server define the class identifier.

Both of these IDs are known as GUIDs, or *globally unique identifiers*. If each developer uses a number to indicate its own OLE servers, how can we be sure that these values are not duplicated? The short answer is that we cannot. The real answer is that a GUID is such a long number (with 16 bytes, or 128 bits, or a number with 38 digits!) that it is almost impossible to come up with two random numbers having the same value. Moreover, programmers should use the specific API call `CoCreateGuid` (directly or through their development environment) to come up with a valid GUID that reflects some system information.

In fact, GUIDs created on machines with network cards are guaranteed to be unique, because network cards contain unique serial numbers that form a base for the GUID creation. GUIDs created on machines with CPU IDs (such as the Pentium III) should also be guaranteed unique, even without a network card. With no unique hardware identifier, GUIDs are unlikely to ever duplicate.

WARNING Besides being careful not to copy the GUID from someone else's program (which can result in two completely different COM objects using the same GUID), you should never make up your own ID by entering a casual sequence of numbers. Windows checks the IDs, and using a casual sequence won't generate a valid ID. An OLE server with an invalid ID is not recognized, and you won't get an error message! Windows also won't include an API or technique to validate a GUID. The risk with creating class or interface IDs by hand is that you could coincidentally duplicate a GUID that is already in use somewhere else in the system. However, to avoid this problem, simply press Ctrl+Shift+G in the Delphi editor, and you will get a new, properly defined, unique GUID.

Delphi defines a `TGUID` data type (in the System unit) to hold these numbers:

```
type
  TGUID = record
    D1: Integer;
    D2: Word;
    D3: Word;
    D4: array [0..7] of Byte;
  end;
```

This structure is actually quite odd but is required by Windows. You can assign a value to a GUID using the standard hexadecimal notation stored inside a string, as in this code fragment:

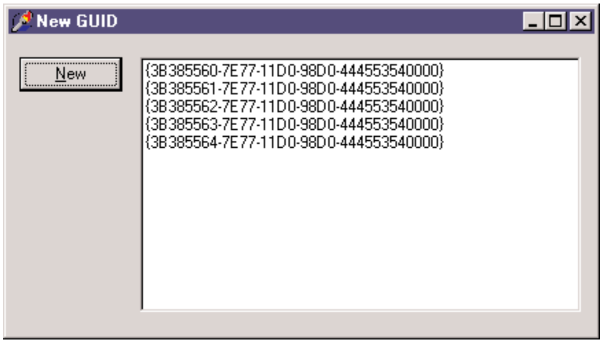
```
const
  Class_ActiveForm1: TGUID = '{1AFA6D61-7B89-11D0-98D0-444553540000}';
```

If you need to generate a GUID manually and not in the Delphi environment, you can simply call the `CoCreateGuid` Windows API function, as demonstrated by the NewGuid example (see Figure 19.1). This example is so simple that I've decided not to list its code.

(You can find the source code for this application, along with the chapter's other examples, in the folder for Chapter 19 on the companion CD.)

FIGURE 19.1:

An example of the GUIDs generated by the NewGuid example. Values depend on my computer and the time I run this program.



```
{3B385560-7E77-11D0-98D0-444553540000}
{3B385561-7E77-11D0-98D0-444553540000}
{3B385562-7E77-11D0-98D0-444553540000}
{3B385563-7E77-11D0-98D0-444553540000}
{3B385564-7E77-11D0-98D0-444553540000}
```

To handle GUIDs, Delphi provides the `GUIDToString` function and the opposite `StringToGUID` function. You can also use the corresponding Windows API functions, such as `StringFromGuid2`, but in this case, you must use the WideString type instead of the string type. Any time OLE is involved, you have to use the WideString type, unless you use Delphi functions that automatically do the required conversion for you. Actually, OLE API functions use the `PWChar` type (pointer to null-terminated arrays of wide characters), but simply casting a WideString to `PWChar` does the trick.

TIP Keep in mind that GUIDs come in different flavors. The two most important types are interface IDs (or IID), which refer to an interface, and class IDs (or CLSID), which refer to a specific object in a server. These two kinds of IDs both use the GUID style.

The Role of Class Factories

When we register the GUID of a COM object in the Registry, we can use a specific API function to create the object, such as the `CreateComObject` API:

```
function CreateComObject (const ClassID: TGUID): IUnknown;
```

This API function will look into the Registry, find the server registering the object with the given GUID, load it, and, if the server is a DLL, call the `DLLGetClassObject` method of the DLL. This is a function every in-process server must provide and export:

```
function DllGetClassObject (const CLSID, IID: TGUID;
  var Obj): HResult; stdcall;
```

This API function receives as parameters the requested class and interface, and it returns an object in its reference parameter. The object returned by this function is a *class factory*.

Now, what is a class factory? As the name suggests, a class factory is an object capable of creating other objects. Each server can have multiple objects. The server exposes the class factory, and the class factory can create one of these various objects. Each object, then, can have multiple interfaces. One of the many advantages of the Delphi simplified approach to COM development is that the system can provide a class factory for us. For this reason, I'm not going to add a class factory to our simple example.

The call to the CreateComObject API doesn't stop at the creation of the class factory, however. After retrieving the class factory, CreateComObject calls the CreateInstance method of the IClassFactory interface. This method creates the requested object and returns it. If no error occurs, this object becomes the return value of the CreateComObject API.

By setting up this mechanism (including the class factory and the DLLGetClassObject call), you make it very simple to create objects. CreateComObject is just a simple function call with a complex behavior behind the scenes. What's great in Delphi is that the complex mechanism is handled for you by the run-time system. So it's time to start looking in detail at how Delphi makes COM really easy to master.

Class Factories and Other Delphi COM Classes

Besides the TComObject class, Delphi includes several other predefined COM classes. We'll use them in the following sections, but here is a list of the most important COM classes of the Delphi VCL:

- TInterfacedObject, defined in the System unit, inherits from TObject and implements the IUnknown interface. It is used only for internal objects.
- TComObject, defined in the ComObj unit, inherits from TObject and implements both the IUnknown interface and the ISupportErrorInfo interface. Unlike TInterfacedObject, this class also has a related class factory.
- TTypedComObject, defined in the ComObj unit, inherits from TComObject and implements the IProvideClassInfo interface (in addition to the IUnknown and ISupportErrorInfo interfaces already implemented by the base class, TComObject).
- TAutoObject, defined in the ComObj unit, inherits from TTypedComObject and implements also the IDispatch interface.
- TActiveXControl, defined in the AxCtrls unit, inherits from TAutoObject and implements several interfaces (IPersistStreamInit, IPersistStorage, IOleObject, and IOleControl, to name just a few).

For each of these classes, Delphi also defines a class factory. The class factory classes form another hierarchy, with the same structure. Their names are TComObjectFactory, TTypedComObjectFactory, TAutoObjectFactory, and TActiveXControlFactory. Class factories are important, and every COM server requires them. Usually we simply use class factories by creating an object in the initialization section of the unit defining the corresponding server object class.

A First COM Server

There is no better way to understand COM than to build a simple COM server hosted by a DLL. A library hosting a COM object is indicated in Delphi as an ActiveX library. For this reason we can start the development of this project by selecting File ➢ New ➢ Other, moving to the ActiveX page, and selecting the ActiveX Library option. This generates a project file I've saved as FirstCom on the companion CD. This is the complete source code:

```
library FirstCom;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

The four functions exported by the DLL are required for COM compliance and are used by the system as follows:

- To access the class library (`DllGetClassObject`)
- To check whether the server has destroyed all its objects and can be unloaded from memory (`DllCanUnloadNow`)
- To add or remove information about the server in the Windows Registry (`DllRegisterServer` and `DllUnregisterServer`)

You generally don't have to implement these functions, because Delphi provides a default implementation in the ComServ unit. For this reason, in the code of our server, we only need to export them.

COM Interfaces and Objects

Now that we have the structure of our COM server in place, we can start developing it. The first step is to write the code of the interface we want to implement in the server. The interface can be very similar to the code of an abstract class, listing all the methods we want to make available from our server. (I have already discussed Object Pascal interfaces in Chapter 3.) Here is the code of a simple interface, which you should add to a separate unit (called NumIntf in the example):

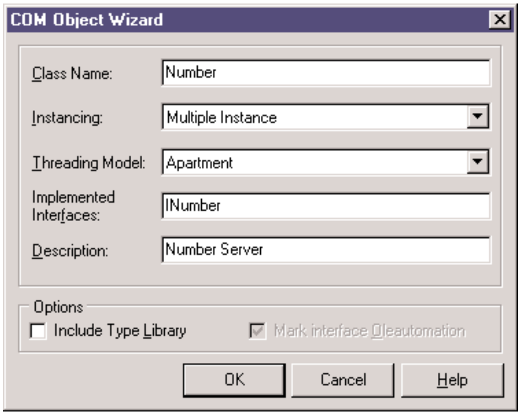
```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

The IID was added to the code by pressing the Ctrl+Shift+G key combination.

After declaring the custom interface, we can add the actual object to the server. To accomplish this, we can use the COM Object Wizard (available in the ActiveX page of the File ➢ New ➢ Other dialog box). You can see this wizard's dialog box in Figure 19.2. Here you should enter the name of the class of the server, the interface you want to implement, and a description. I've disabled the generation of the type library to avoid introducing too many topics at once. You should also choose an instancing and a threading model, as described in the related sidebar.

FIGURE 19.2:

The COM Object Wizard



COM Object Wizard

Class Name: Number

Instancing: Multiple Instance

Threading Model: Apartment

Implemented Interfaces: INumber

Description: Number Server

Options

☐ Include Type Library ☑ Mark interface Oleautomation

The code generated by the COM Object Wizard is actually quite simple. The interface contains the definition of the class to fill with methods and data:

```
type
  TNumber = class(TComObject, INumber)
    protected
      {Declare INumber methods here}
    end;
```

The server class inherits from the TComObject class, which I discussed in the last section. In the code generated by the wizard, after the server class comes the definition of the GUID for the server:

```
const
  Class_Number: TGUID = '{5B2EF181-3AAE-11D3-B9F1-00000100A27B}';
```

Finally, there is some code in the initialization section (which uses most of the options we've set up in the wizard's dialog box):

```
initialization
  TComObjectFactory.Create(ComServer, TNumber, Class_Number, 'Number',
    'Number Server', ciMultiInstance, tmApartment);
```

This code creates an object of the TComObjectFactory class, passing as parameters the global ComServer object, a class reference to the class we've just defined, the GUID for the class, the server name, the server description, and the instancing and threading models we want to use.

The global ComServer object, defined in the ComServ unit, is a manager of the class factories available in the server library. It uses its own ForEachFactory method to look for the class supporting a given COM object request, and it keeps track of the number of allocated objects. As we've already seen, in fact, the ComServ unit implements the functions required by the DLL to be a COM library.

Having examined the source code generated by the wizard, we can now complete it by adding to the TNumber class the methods required for implementing the INumber interface. First, write the declaration of the methods:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
  end;
```

At this point, simply activate class completion by pressing Shift+Ctrl+C and fill the methods with the proper code. This is so straightforward that I'm not going to list it here; you can find the source code on the companion CD.

COM Instancing and Threading Models

When you create a COM server, you should choose a proper instancing and threading model, which can significantly affect the behavior of the COM server.

Instancing affects only out-of-process servers (any COM server in a separate executable file, rather than a DLL) and can assume three values:

Multiple indicates that when several client applications require the COM object, the system starts multiple instances of the server.

Single indicates that, even when several client applications require the COM object, there is only one instance of the server application; it creates multiple internal objects to service the requests.

Internal indicates that the object can only be created inside the server; client applications cannot ask for one.

The second decision relates to the thread support of the COM object, which is valid for in-process servers only (DLLs). The threading model is a joint decision of the client and the server application: if both sides agree on one model, it is used for the connection. If no agreement is found, COM can still set up a connection using marshaling, which can slow down the operations. And keep in mind that a server must not only publish its threading model in the Registry (as a result of setting the option in the wizard); it must also follow the rules for that threading model in the code. Here are the key highlights of the various threading models:

The Single model indicates no real support for threads. The requests reaching the COM server are serialized, so that the client can perform one operation at a time.

The Apartment model, or "single-threaded Apartment" indicates that only the thread that created the object can call its methods. This means that the requests for each server object are serialized, but other objects of the same server can receive requests at the same time. For this reason, the server object must take extra care only to access global data of the server (using critical sections, mutexes, or some other synchronization techniques). This is the threading model generally used for ActiveX controls inside Internet Explorer.

The Free model, or "multi-threaded Apartment" indicates that the client has no restrictions, which means that multiple threads can use the same object at the same time. For this reason, every method of every object must protect itself and the nonlocal data it uses against multiple simultaneous calls. This threading model is more complex for a

Continued on next page

server to support than the Single and Apartment models, because even access to the object's own instance data must be handled with thread-safe care.

The fourth option, Both indicates that the server object supports both the Apartment model and the Free model.

The final option, Neutral was introduced in Windows 2000 and is available only under COM+. It indicates that multiple clients can call the object on different threads at the same time, but COM guarantees you that the same method is not invoked twice at the same time. Guarding for concurrent access to the data of the objet is required. Under COM, it is mapped to the Apartment model.

Initializing the COM Object

If you look back at the definition of the TComObject class, you will notice it has a nonvirtual constructor. (Actually, it has multiple nonvirtual constructors, which I've omitted from the listing.) Each TComObject constructor calls the virtual Initialize method. For this reason, if you want to customize the creation of an object and then initialize it, you should not define a new constructor (which will never be called). What you should do is override its Initialize method, as I've done in the TNumber class. Here is the final version of this class:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
    procedure Initialize; override;
    destructor Destroy; override;
  end;
```

As you can see, I've also overridden the destructor of the class, because I wanted to test the automatic destruction of the COM objects provided by Delphi. Here is the code for this pseudoconstructor and the destructor:

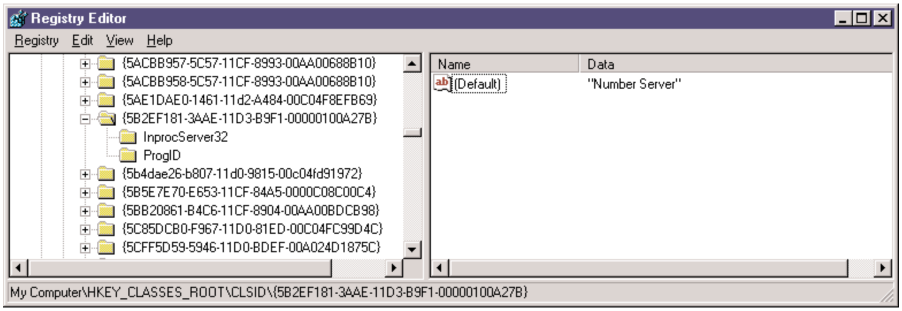
```
procedure TNumber.Initialize;
begin
  inherited;
  fValue := 10;
end;
```

```
destructor TNumber.Destroy;
begin
  inherited;
  MessageBox (0, 'Object Destroyed', 'TDLLNumber', mb_OK); // API call
end;
```

In the first method, calling the inherited version is good practice, even though the `TComObject.Initialize` method has no code in this version of Delphi. The destructor, instead, must call the base class version. This is the code required to make our COM object work properly and to let us know when an object is actually destroyed.

Testing the COM Server

Now that we've finished writing our COM server object, we can register and use it. Simply compile its code and then use the Run ➢ Register ActiveX Server menu command in Delphi. You do this to register the server on your own machine, with the results you can see in Figure 19.3.



```
{5ACBB957-5C57-11CF-8993-00AA00688B10}
{5ACBB958-5C57-11CF-8993-00AA00688B10}
{5AE1DAE0-1461-11d2-A484-00C04F8EFB69}
{5B2EF181-3AAE-11D3-B9F1-00000100A27B}
    InprocServer32
    ProgID
{5b4dae26-b807-11d0-9815-00c04fd91972}
{5B5E7E70-E653-11CF-84A5-0000C08C00C4}
{5BB20861-B4C6-11CF-8904-00AA00BDCB98}
{5C85DCB0-F967-11D0-81ED-00C04FC99D4C}
{5CFF5D59-5946-11D0-BDEF-00A024D1875C}
```

Name	Data
(Default)	"Number Server"

My Computer\HKEY_CLASSES_ROOT\CLSID\{5B2EF181-3AAE-11D3-B9F1-00000100A27B}

FIGURE 19.3:

The new registered server in Windows RegEdit

When you distribute this server, you should install it on the client computers. To accomplish this, you can write a REG file to install the server in the Registry. However, this is not really the best approach, because the server already includes a function you can activate to register the server. This function can be activated by the Delphi environment, as we've seen, or in a few other ways:

- You can pass the COM server DLL as a command-line parameter to Microsoft's `RegSvr32.exe` program, found in the `\Windows\System` directory.
- You can use the similar `TRegSvr.exe` demo program that ships with Delphi. (The compiled version is in the `\Bin` directory, and its source code is in the `\Demos\ActiveX` directory.)
- You can let an installation builder program call the registration function of the server.

Having registered the server, we can now turn to the client side of our example. This time, the example is called TestCom and is stored in a separate directory. In fact, the program loads the server DLL through the OLE/COM mechanism, thanks to the server information present in the Registry, so it's not necessary for the client to know which directory the server resides in.

The form displayed by this program is very similar to the one we've used to test the object inside the DLL. In the client program, you must include the source code file with the interface and redeclare the COM server GUID. Of course, the code of the program's `FormCreate` method should be updated to create the required COM objects. The program starts with all the buttons disabled (at design time), and it enables them only after an object has been created. This way, if an exception is raised while creating one of the objects, the buttons related to the object won't be enabled:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // create first object
  Num1 := CreateComObject (Class_Number) as INumber;
  Num1.SetValue (SpinEdit1.Value);
  Label1.Caption := 'Num1: ' + IntToStr (Num1.GetValue);
  Button1.Enabled := True;
  Button2.Enabled := True;

  // create second object
  Num2 := CreateComObject (Class_Number) as INumber;
  Label2.Caption := 'Num2: ' + IntToStr (Num2.GetValue);
  Button3.Enabled := True;
  Button4.Enabled := True;
end;
```

Notice in particular the call to `CreateComObject` and the following `as` cast. The API call starts the COM object-construction mechanism I've already described in detail. This call also dynamically loads the server DLL. The return value is an `IUnknown` object. This object must be converted to the proper interface type before assigning it to the `Num1` and `Num2` fields, which now have the interface type `INumber` as their data type:

```
type
  TForm1 = class(TForm)
    ...
  private
    Num1, Num2 : INumber;
```

WARNING To downcast an interface to the actual type, *always* use the `as` cast, which for interfaces performs a `QueryInterface` call behind the scenes. This provides some protection, because it raises an exception if the interface you are casting to is not supported by the given object. In the case of interfaces, the `as` cast is the only way to *extract* an interface from another interface. If you write a plain cast of the form `INumber(CreateComObject(Class_Number))`, the program will crash, even if the cast seems to make sense, as in the case above. Casting an interface pointer to another interface pointer is an error. Period. Never do it.

In Figure 19.4, you can see the output of this test program, which is very similar to the previous version. Notice that this time, `Num2` shows the initial value of the object at start-up, as set up in its `Initialize` method. Notice also that I've added one more button, which creates a third temporary COM object:

```
procedure TForm1.Button5Click(Sender: TObject);
var
  Num3: INumber;
begin
  // create a new temporary COM object
  Num3 := CreateComObject (Class_Number) as INumber;
  Num3.SetValue (100);
  Num3.Increase;
  ShowMessage ('Num3: ' + IntToStr (Num3.GetValue));
end;
```

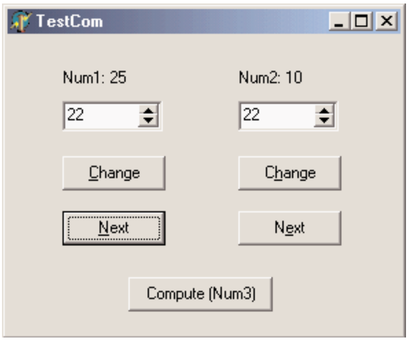


FIGURE 19.4: The output of the TestCom example, a COM client

Pressing this button, you simply get the value of the number following 100. To see why I added this method to the example, you need to press the button a second time, after the message showing the result. Now you get a second message, indicating that the object has been destroyed. This demonstrates that simply letting an interface object go out of scope automatically calls the object's `Release` method, decreases the object's reference count, and destroys the object if its reference count reaches zero. Chapter 3 described this reference-counting mechanism in more detail.

The same happens to the other two objects as soon as the program terminates. Even if the program doesn't explicitly destroy the two objects in the FormDestroy method, they are indeed destroyed, as the message shown by their Destroy destructor clearly demonstrates. This happens because they were declared to be of an interface type, and Delphi is going to use reference counting for them.

Using Interface Properties

As a further small step, we can extend the example by adding a property to the INumber interface. When you add a property to an interface, you indicate the data type and then the read and write directives. You can have read-only or write-only properties, but the read and write clauses must always refer to a method because interfaces don't hold anything else but methods.

Here is the updated interface, which is part of the PropCom example:

```
type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer read GetValue write SetValue;
    procedure Increase; stdcall;
  end;
```

I've given this interface a new name and, what's even more important, a new interface ID. I could have inherited the new interface type from the previous one, but this would have provided no real advantage. COM by itself doesn't really support inheritance, and from the perspective of COM, all interfaces are different simply because they have different interface IDs. Needless to say, in Delphi we can use inheritance to improve the structure of the code of the interfaces and of the server objects implementing them.

In the PropCom example, I've updated the server class declaration simply by writing:

```
type
  TDllNumber = class (TComObject, INumberProp)
  ...
```

This class also has a new server object ID. The client program, also on the CD, can now simply use the Value property instead of the SetValue and GetValue methods. Here is a small excerpt from the FormCreate method:

```
Num1 := CreateComObject (Class_NumPropServer) as INumberProp;
Num1.Value := SpinEdit1.Value;
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

The difference between using methods and properties for an interface is only syntactical, because interface properties cannot access private data as class properties can. By using properties, we can make the code a little more readable.

Calling Virtual Methods

We've built a couple of examples based on COM, but you might still feel uncomfortable with the idea of a program calling methods of objects that are created within a DLL. How is this possible if those methods are not exported by the DLL? The COM server, the DLL, creates an object and returns it to the calling application. By doing this, the DLL creates an object with a *virtual method table* (VMT). (Remember that all the interface methods are virtual by default.)

Because every object embeds a pointer to its VMT, the main program receives an object, and also a way to work on it, by calling its virtual methods. The main program doesn't need to know the memory address of those methods, because the objects know it, exactly as they do with a polymorphic call. But COM is even more powerful than this: you don't even have to know which programming language was used to create the object, provided its VMT follows the standard dictated by COM.

TIP The COM-compatible VMT implies also a strange effect. The method names are not important, provided their address is in the proper position in the VMT. This is why you can map a method of an interface to an actual function implementing it.

To sum things up, we can say that COM provides a language-independent binary standard for objects. The objects you share among modules are compiled, and their VMT has a particular structure, determined by COM and not by the development environment you've used.

Windows Shell Programming

In the last section, we built a fully standard COM object, packaged it as an in-process server, and used it from a standard client. However, the COM interface we implemented was a custom interface we'd built. Now we can try to build clients and servers related to the Windows shell interfaces, which are all based on COM. The original Windows API was basically a collection of functions, but all the most recent APIs are generally based on COM.

The following sections use some existing servers that are part of the Windows shell; we'll write a client application and use the COM servers provided by the system. This case illustrates the difference from the traditional use of the Windows API calls. I'm also going to write some COM servers to be used by the Windows system, particularly the Explorer. This case illustrates the difference from the traditional development of a callback function invoked by the system.

Creating Shortcuts

One of the simplest shell interfaces we can use in a client application is the `IShellLink` interface. This interface relates to Windows shortcuts and allows programmers to access the information of an existing shortcut or to create a new one. In the ShCut example on the CD-ROM, I'm going to create various types of shortcuts, all referring to the program itself. Of course, once you understand how to do this, you can easily extend the example and create shortcuts for any program or file.

The example has an edit box for the name of the shortcut, a few check boxes, and two buttons. When the Create button is pressed, the text in the edit box is used as the name of a new shortcut, which is placed in the current directory, on the desktop, or in the Start menu. These options are not exclusive; a user can create multiple shortcuts at once.

The most important code is at the very beginning of this method. The `CreateComObject` call creates a system object, as indicated by the GUID passed as a parameter. The result of this call (which is an `IUnknown` interface) is converted both to an `IShellLink` interface and to an `IPersistFile` interface:

```
uses
  ComObj, ActiveX, ShlObj, Registry;

procedure TForm1.Button1Click(Sender: TObject);
var
  AnObj: IUnknown;
  ShLink: IShellLink;
  PFile: IPersistFile;
  FileName: string;
  WFileName: WideString;
  Reg: TRegIniFile;
begin
  // access the two interfaces of the object
  AnObj := CreateComObject (CLSID_ShellLink);
  ShLink := AnObj as IShellLink;
  PFile := AnObj as IPersistFile;
```

Actually, we could have written the last three lines of code above using this shorter notation:

```
ShLink := CreateComObject (CLSID_ShellLink) as IShellLink;
PFile := ShLink as IPersistFile;
```

If you look at similar examples built in other languages, you'll notice that to access the IPersistFile interface, the programs use custom calls to the QueryInterface method. The two as expressions basically call QueryInterface for us.

Once we have the IShellLink interface, we can call some of its methods, such as SetPath and SetWorkingDirectory:

```
// get the name of the application file
FileName := ParamStr (0);
// set the link properties
ShLink.SetPath (PChar (FileName));
ShLink.SetWorkingDirectory (PChar (ExtractFilePath (FileName)));
```

Once we've set up the shell link object, we have to save it, depending on the status of the three check boxes, calling the Save method of the IPersistFile interface of the object. The simplest version is the one used to save the link in the current directory:

```
// save the file in the current dir
if cbDir.Checked then
begin
  // using a WideString
  WFileName := ExtractFilePath (FileName) + EditName.Text + '.lnk';
  PFile.Save (PWChar (WFileName), False);
end;
```

The call to the Save method (which creates the physical LNK file) requires a "pointer to wide char" parameter. The simplest way to obtain this is to declare a long string and then cast it to a PWChar. Do not try casting a plain string to PWChar—the compiler will emit a warning and the program won't work!

To create the shortcut on the desktop or in the Start menu, we should first determine the corresponding system folder by looking up the proper value in the Registry. By writing the program this way, we ensure it will work on different versions of Windows and on localized versions as well. Here is the source code for the last two check boxes:

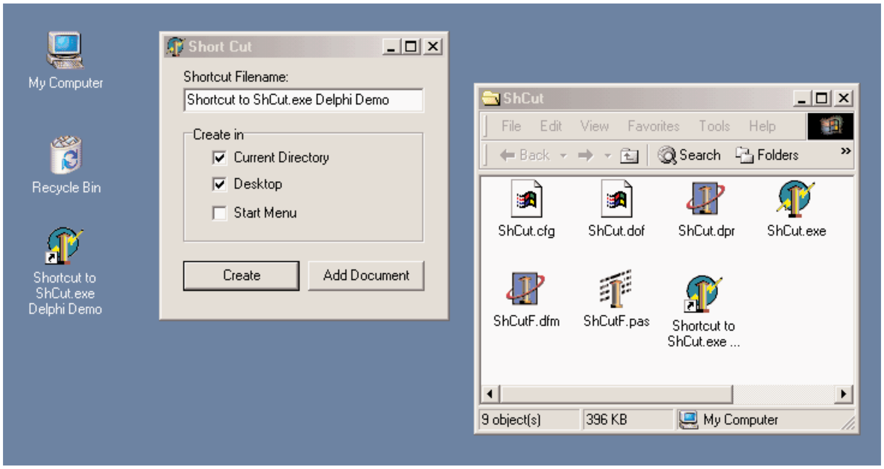
```
// save on the desktop
if cbDesktop.Checked then
begin
  Reg := TRegIniFile.Create(
    'Software\MicroSoft\Windows\CurrentVersion\Explorer');
  WFileName := Reg.ReadString ('Shell Folders', 'Desktop', '') +
    '\' + EditName.Text + '.lnk';
  Reg.Free;
  PFile.Save (PWChar (WFileName), False);
end;
// save in the Start Menu
if cbStartMenu.Checked then
begin
```

```
    Reg := TRegIniFile.Create(
      'Software\MicroSoft\Windows\CurrentVersion\Explorer');
    WFileName := Reg.ReadString ('Shell Folders', 'Start Menu', '') +
      '\' + EditName.Text + '.lnk';
    Reg.Free;
    PFile.Save (PWChar (WFileName), False);
  end;
```

To look up the information in the Registry, I've used the `TRegIniFile` class, although there are other related classes in the VCL, such as the `TRegistry` class. The effect of running this program and pressing the button is that Windows will add a new link in the directory of the project, on the desktop, or in the Start menu. You can see an example of the program in Figure 19.5.

FIGURE 19.5:

The simple user interface of the ShCut example, and two shortcuts created with it in the project folder and on the desktop



Shortcut Filename: Shortcut to ShCut.exe Delphi Demo

Create in
☑ Current Directory
☑ Desktop
□ Start Menu

9 object(s) | 396 KB | My Computer

Using Shell APIs and Objects

As an extra feature, the program can also add a new document to the list of recently used ones, calling the `SHAddToRecentDocs` method:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  ProjectFile: string;
begin
  ProjectFile := ChangeFileExt (ParamStr (0), '.dpr');
  SHAddToRecentDocs (SHARD_PATH, PChar(ProjectFile));
end;
```

This has very little to do with COM, and I've added it to the example only to highlight that there is a very large number of shell-related APIs, available in the ShlObj unit, besides the original and more limited ShellApi unit.

Another example, available in the source code for this chapter and called FindFolders, highlights the use of another *plain* (non-COM) shell function, `SHBrowseForFolder`. In the example you can see the following code:

```
procedure TForm1.btnBrowseClick(Sender: TObject);
var
  bi: TBrowseInfo;
  pidl: pItemIdList;
  strpath, displayname: string;
begin
  SetLength (displayname, 100);

  bi.hwndOwner := Handle;
  bi.pidlRoot := nil;
  bi.pszDisplayName := pChar (displayname);
  bi.lpszTitle := 'Select a folder';
  bi.ulFlags := bif_StatusText;
  bi.lpfn := nil;
  bi.lParam := 0;

  pidl := SHBrowseForFolder (bi);

  SetLength (strPath, 100);
  SHGetPathFromIdList (pidl, PChar(strPath));
  Edit1.Text := strPath;
end;
```

The FindFolders example even shows some Delphi-specific APIs to interact with files and folders (available also on Linux) including `SelectDirectory`, which has the same effect of `SHBrowseForFolder` but a different user interface. The example also uses the `DirectoryExists` and `ForceDirectories` functions, available in the FileCtrl unit. You can see how they are used by looking in the source code of the example.

NOTE Notice that in Delphi 6, some of the Shell API is also encapsulated in the sample ShellListView, ShellTreeView, and ShellComboBox controls. I've used these controls in a few examples throughout the book, including the DirDemo example of Chapter 18, "Writing Database Components."

The "To-Do File" Application

As a second example of integrating a Delphi program with the system shell, I've tried to write a simple real-world application that uses file dragging and a context menu handler. I'll start with the file dragging first, because this will actually introduce some of the techniques used by the context menu handler.

As I mentioned, this application is actually useful; you can use it to create a sort of "to-do list." It is based on a Paradox table that stores filenames and notes about the files. The form of the application has a DBGrid component showing only a single column containing the filenames and a memo control hosting the notes related to the current file. You can see this form at design time in Figure 19.6.

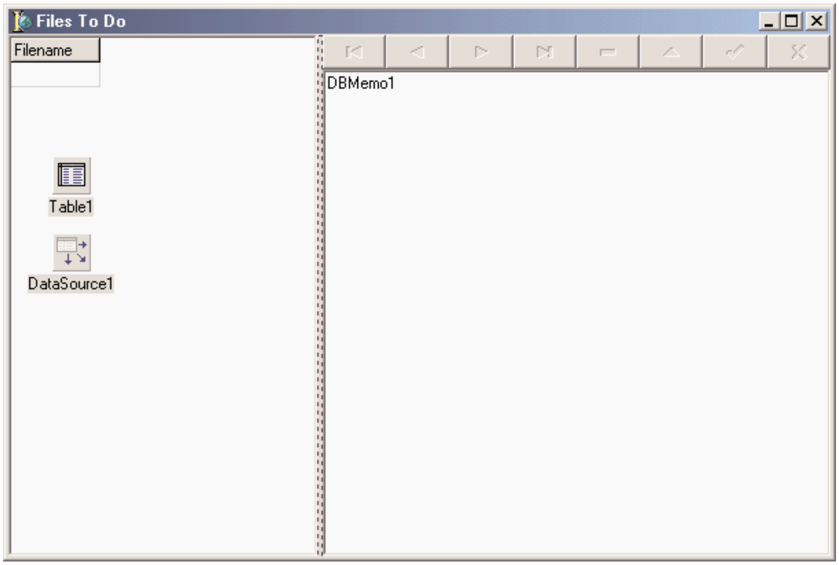


FIGURE 19.6: The form of the ToDoFile example at design time

TIP Using a single-column DBGrid is the only way in Delphi to show a list of the available records in a list-box format. The alternative, of course, is to fill a list box with custom code and then manually navigate in the database table when the selection in the list box changes. This manual approach is, of course, less efficient when we have many records, because the program needs to scan them all to fill the list box, while the DBGrid loads only the record it currently displays.

Notice that the navigator component has no "New Record" button, and the DBGrid is set up as a read-only component. In fact, users should not be able to create new records except by dragging a file onto the form, and they're not allowed to change the filename field in any way (except by deleting it). All the user can do is edit the notes field, entering a description of the operations to be done on the file.

Creating the Database

To create the database table for this example, I've used the `FieldDefs` property to define the structure and set the `StoreDefs` property to True to save the table definition along with the form DFM file. The table has two fields, a string field called Filename and a memo field called Notes. Of course, you can also create the table at design time, using the table component's local menu. The program, however, calls the `CreateTable` method in the `OnCreate` event handler, unless this has already been done:

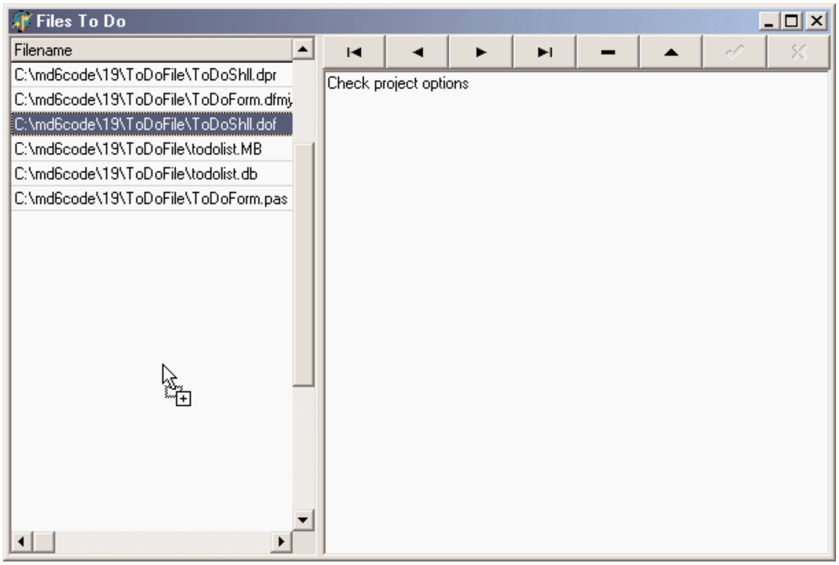
```
procedure TToDoFileForm.FormCreate(Sender: TObject);
begin
  // eventually create the table
  if not Table1.Exists then
    Table1.CreateTable;
  // activate the table
  Table1.Activate;
  // accept dragging to the form
  DragAcceptFiles (Handle, True);
end;
```

Dragging Files to the Form

As you can see in the listing above, the form initialization code also registers the window with the system as a file-dragging target, by calling the `DragAcceptFiles` Windows API function. As a result, the application's cursor changes to the typical "drag accept" icon when a file is dragged over it. You can see an example of this cursor in Figure 19.7.

FIGURE 19.7:

The drag-accept cursor displayed by the ToDoFile application as a user drags a file over it



Filename
C:\md6code\19\ToDoFile\ToDoShll.dpr
C:\md6code\19\ToDoFile\ToDoForm.dfm
C:\md6code\19\ToDoFile\ToDoShll.dof
C:\md6code\19\ToDoFile\todolist.MB
C:\md6code\19\ToDoFile\todolist.db
C:\md6code\19\ToDoFile\ToDoForm.pas

Check project options

When a file-dragging operation is performed, the system sends the window a wm_DropFiles message. This message passes (among its other parameters) a handle to a file-drop structure from which you can extract information by using the DragQueryFile API function. When this API function is called with the $FFFFFFFF parameter, it returns the number of files dragged to the window; when it is called with a numeric parameter, it fills a buffer with the name of that file. For this reason, the code of a wm_DropFiles message handler gets the number of files first and then loops for each of the files, as the following listing demonstrates:

```
procedure TToDoFileForm.DropFiles(var Msg: TWmDropFiles);
var
  nFiles, I: Integer;
  Filename: string;
begin
  // get the number of dropped files
  nFiles := DragQueryFile (Msg.Drop, $FFFFFFFF, nil, 0);
  // for each file
  try
    for I := 0 to nFiles - 1 do
    begin
      // allocate memory
      SetLength (Filename, 80);
      // read the file name
      DragQueryFile (Msg.Drop, I, PChar (Filename), 80);
      // normalize file
      Filename := PChar (Filename);
      // add a new record
      Table1.InsertRecord ([Filename, '']);
    end;
  finally
    DragFinish (Msg.Drop);
  end;
  // open the (last) record in edit mode
  Table1.Edit;
  // move the input focus to the memo
  DBMemo1.SetFocus;
end;
```

As you can see in the preceding code, for every new file, the program inserts a new record with the corresponding filename and an empty field for the notes. Then, for the last file being dragged, the program opens the record in edit mode and moves the focus to the memo control, so that a user can fill the notes for the file.

Creating a Context-Menu Handler

Now that we have the base program running, we can add a shell extension to the system to let the user simply select a file and "send" it to the application without having to do the dragging operation, which is not always handy when there are many programs running. A context-menu extension is one of the available Windows shell extensions and is activated every time a user right-clicks a file in the Windows Explorer.

Technically, a context menu is a COM server exposing an internal object that is going to be created and used by the system. A context-menu COM object must implement two different interfaces, `IContextMenu` and `IShellExtInit`. The first interface defines specific actions for the context menu, such as defining the number of menu items to add and their text, while the second interface defines a way to access the file or files the user is operating on. This is the resulting definition of the COM server object class:

```
type
  TToDoMenu = class(TComObject, IUnknown, IContextMenu, IShellExtInit)
  private
    fFileName: string;
  protected
    {Declare IContextMenu methods here}
    function QueryContextMenu(Menu: HMENU; indexMenu,
      idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(
      var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
    {Declare IShellExtInit methods here}
    function IShellExtInit.Initialize = InitShellExt;
    function InitShellExt (pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
  end;
```

Notice that the class implements the `Initialize` method of the `IShellExtInit` interface with a differently named method, `InitShellExt`. The reason is that I wanted to avoid confusion with the `Initialize` method of the `TComObject` base class, which is the hook we have to initialize the object, as described earlier in this chapter. Let's examine the `InitShellExt` method first; it is definitely the most complex one:

```
function TToDoMenu.InitShellExt(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
var
  medium: TStgMedium;
  fe: TFormatEtc;
begin
  Result := E_FAIL;
```

```
    // check if the lpdobj pointer is nil
    if Assigned (lpdobj) then
    begin
      with fe do
      begin
        cfFormat := CF_HDROP;
        ptd := nil;
        dwAspect := DVASPECT_CONTENT;
        lindex := -1;
        tymed := TYMED_HGLOBAL;
      end;
      // transform the lpdobj data to a storage medium structure
      Result := lpdobj.GetData(fe, medium);
      if not Failed (Result) then
      begin
        // check if only one file is selected
        if DragQueryFile (medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
        begin
          SetLength (fFileName, 1000);
          DragQueryFile (medium.hGlobal, 0, PChar (fFileName), 1000);
          // realign string
          fFileName := PChar (fFileName);
          Result := NOERROR;
        end
        else
          Result := E_FAIL;
      end;
      ReleaseStgMedium(medium);
    end;
  end;
```

The initial portion of the method transforms the pointer to the IDataObject interface, which we receive as a parameter, into the same data structure used in a file drop operation, so that we can read the file information by using the DragQueryFile function again. This complex way of coding is actually the simplest one you can use! At the end of this operation, we have the value of the filename. Any selection of multiple files is not accepted.

We can now look at the methods of the IContextMenu interface. The first, QueryContextMenu, is used to add new items to the local menu of the file. In this case, we add a new menu item (calling the InsertMenu API function) only if the ToDoFile application is running. We can determine this by searching for a window corresponding to the TToDoFileForm class, which should be unique in the system. The result of the function is the number of items added to the menu:

```
function TToDoMenu.QueryContextMenu(Menu: HMENU;
  indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
```

```
    // add entry only if the program is running
    if FindWindow ('TToDoFileForm', nil) <> 0 then
    begin
      // add a new item to context menu
      InsertMenu (Menu, indexMenu, MF_STRING or MF_BYPOSITION, idCmdFirst,
        'Send to ToDoFile');
      // return the number of menu items added
      Result := 1;
    end
    else
      Result := 0;
  end;
```

Now that items have been added to the menu, a user can select them. While he or she moves over the items, a descriptive message is displayed in the status bar of the Windows Explorer. The menu ID (idCmd) we receive in the GetCommandString method is simply the relative number, starting with zero, of the items we have added to the menu. When the cursor is over an item, we simply copy a string with its description to the buffer provided by the system:

```
function TToDoMenu.GetCommandString(idCmd, uType: UINT;
  pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  if idCmd = 0 then
  begin
    // return help string for menu item
    strCopy (pszName, 'Add file to the ToDoFile database');
    Result := NOERROR;
  end
  else
    Result := E_INVALIDARG;
end;
```

The final step is the operation to do once a menu item is actually selected. The InvokeCommand method receives a pointer to a structure holding the request. This method follows a standard pattern of first checking that the request is valid by looking at the two 16-bit words of the lpici.lpVerb value. After these preliminary (but required) steps, we check the value to see which menu item was activated; or, if the context menu has only one item, as in this case, we simply test for a value of zero. The following is the skeleton of the code, before we add the specific action:

```
function TToDoMenu.InvokeCommand (var lpici: TCMInvokeCommandInfo): HResult;
begin
  Result := NOERROR;
  // make sure we are not being called by an application
  if HiWord(Integer(lpici.lpVerb)) <> 0 then
  begin
    Result := E_FAIL;
    Exit;
    end;
    // make sure we aren't being passed an invalid argument number
    if LoWord(lpici.lpVerb) > 0 then
    begin
      Result := E_INVALIDARG;
      Exit;
    end;
    // execute the command specified by lpici.lpVerb
    if LoWord(lpici.lpVerb) = 0 then
    begin
      // actual code still missing here
    end
  end;
```

Sending Data to Another Application with *wm_CopyData*

Because we have the filename the user is operating on, all we have to do in the context-menu handler is send this name to the main form of the ToDoFile application. The problem is that the context-menu handler DLL runs in the Windows Explorer process, so it cannot send the value of a memory pointer to another process. This would simply be useless; as in Win32, different applications have separate memory address spaces.

We saw in the last chapter that one way to share data among applications is to use a memory-mapped file. Another technique, which is actually better in this case, is to use the wm_CopyData message. This is a special Windows message, which can be used to send a memory buffer to another application: Windows will resolve all the memory conversion problems for us. A program basically fills the CopyDataStruct data structure with the data and indicates its length, and then must use the SendMessage API to forward it to a destination window. For this reason we need to use FindWindow again to get the handle of the main window of the ToDoFile application. Here is the rest of the code of the InvokeCommand method:

```
  var
    hwnd: THandle;
    cds: CopyDataStruct;
  begin
    ...
    if LoWord(lpici.lpVerb) = 0 then
    begin
      // get the handle of the window
      hwnd := FindWindow ('TToDoFileForm', nil);
      if hwnd <> 0 then
      begin
        // prepare the data to copy
        cds.dwData := 0;
        cds.cbData := length (fFileName);
        cds.lpData := PChar (fFileName);
      // activate the destination window
      SetForegroundWindow (hwnd);
      // send the data
      SendMessage (hwnd, wm_CopyData, lpici.hWnd, Integer (@cds));
    end;
  end;
```

NOTE Before sending the data, we must activate the destination window by calling the `SetForegroundWindow` API. This is necessary because we are going to activate a window that was created by another thread, something Windows doesn't normally do. Notice also that if you write this call in the ToDoFile application as it receives the `wm_CopyData` message, it will produce no effect at all.

As the context-menu handler sends data to it, the application has to be extended to handle the `wm_CopyData` message. In this event handler, we receive the same structure we sent for the other side, although between the send operation done by the context-menu handler and the receive operation done by the application. Windows takes care of mapping the data properly to the other address space. As a result, extracting the filename is actually very simple, but keep in mind that this is so only because Windows does a lot of work behind the scenes. Using a plain Windows message other than `wm_CopyData` will never work!

Here is the code I've added to the form of the ToDoFile application. It does several things: It restores the application if it was minimized, retrieves the name of the file, inserts a new record in the database table, copies the filename, and moves the focus to the memo control once more.

```
procedure TToDoFileForm.CopyData(var Msg: TWmCopyData);
var
  Filename: string;
begin
  // restore the window if minimized
  if IsIconic (Application.Handle) then
    Application.Restore;

  // extract the filename from the data
  Filename := PChar (Msg.CopyDataStruct.lpData);
  // insert a new record
  Table1.Insert;
  // set up the file name
  Table1.FieldByName ('Filename').AsString := Filename;
  // move the input focus to the memo
  DBMemo1.SetFocus;
end;
```

Registering the Shell Extension

After writing this shell extension, we must register it. With the usual Run ➢ Register ActiveX Server command, we can register the server in the system, but we still have to provide some extra information to register it as a shell extension, in this case for any type of file. There are several approaches: you can edit the Registry manually, you can write a REG file, or you can add registration information right into the COM server library, which is my preferred approach. In a Delphi COM server, the default registration takes place in the `TComObjectFactory` class, when the `UpdateRegistry` method is executed. We can modify the default registration by inheriting a class from the standard class factory class and overriding this method:

```
type
  TToDoMenuFactory = class (TComObjectFactory)
  public
    procedure UpdateRegistry (Register: Boolean); override;
  end;
```

In this method, we should either add the entry in the Registry or delete it, depending on the value of the Boolean parameter:

```
procedure TToDoMenuFactory.UpdateRegistry(Register: Boolean);
var
  Reg: TRegistry;
begin
  inherited UpdateRegistry (Register);

  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_CLASSES_ROOT;
  try
    if Register then
      if Reg.OpenKey('\*\ShellEx\ContextMenuHandlers\ToDo', True) then
        Reg.WriteString('', GUIDToString(Class_ToDoMenuMenu))
    else
      if Reg.OpenKey('\*\ShellEx\ContextMenuHandlers\ToDo', False) then
        Reg.DeleteKey ('\*\ShellEx\ContextMenuHandlers\ToDo');
  finally
    Reg.CloseKey;
    Reg.Free;
  end;
end;
```

WARNING I've checked this code under Windows 2000, but I'm not completely sure it works also on Windows 98/Me, as the shell portion of the registry has been subject to subtle changes among different versions of Windows.

In the initialization section of the COM object unit, we also need to create a new global object of this class instead of the base class factory class:

```
initialization
  TToDoMenuFactory.Create (ComServer, TToDoMenu, Class_ToDoMenuMenu,
    'ToDoMenu', 'ToDoMenu Shell Extension', ciMultiInstance, tmApartment);
```

Now you can simply register the server and set it up as a context-menu handler by using the Delphi Run ➢ Register ActiveX Server menu command, the RegSrv32 application, or most of the tools used to create installation programs.

What’s Next?

In this chapter I have discussed the foundations of Microsoft’s COM technology. We’ve seen how Delphi supports COM and built a few simple servers. In the second part of the chapter, we’ve spend some time discussing the COM-based Shell API and the development of a shell extension.

The next chapter opens up COM to its higher-level techniques, covering Automation, Documents, and ActiveX Controls. Now that we know the foundations, exploring these COM-technologies will definitely be simpler, although we won’t delve into the low-level details of these technologies.

CHAPTER 20

From Automation to COM+

- OLE Automation
- Creating and using Automation servers
- Using type libraries
- Automating office programs
- The OLE Container component
- Building an ActiveX and an ActiveForm
- Introducing COM+

After the last chapter, which was devoted to the foundations of Microsoft's COM architecture, it is time to look into some of the actual high-level Windows programming techniques based on COM. We'll start by discussing Automation and the role of type libraries. Also, we'll see how to work properly with Delphi data types in Automation servers and clients.

Later, we'll focus on the use of the Automation support provided by Microsoft Office applications, made simple thanks to the ready-to-use components that embed Office server programs and documents. In the final part of the chapter, we'll explore the use of embedded objects, with the OleContainer component, and the development of OLE controls or ActiveX controls.

I'll also introduce stateless COM (MTS and COM+) technologies and a few other advanced ideas. But let's begin with more foundational material.

OLE Automation

In the last chapter, we saw that you can use COM to let an executable file and a library share objects, and that this can be used to interact with the Windows shell. Most of the time, however, users want applications that can talk to each other. One of the approaches you can use for this goal is OLE Automation. After presenting a couple of examples that use custom interfaces based on type libraries, I'll cover the development of Word and Excel OLE controllers, showing how to transfer database information to those applications.

NOTE The current Microsoft documentation uses the term *Automation* instead of *OLE Automation,* and uses the terms *active document* and *compound document* instead of *OLE Document*. This book uses this new terminology along with the older "OLE" terminology incorporated into many Delphi component names and other identifiers.

In Windows, applications don't live in separate worlds; users often want them to interact. The Clipboard and DDE offer a simple way for applications to interact, as users can copy and paste data between applications. However, more and more programs offer an OLE Automation interface to let other programs drive them. Beyond the obvious advantage of programmed automation compared to manual user operations, these interfaces are completely language-neutral, so you can use Delphi, C++, Visual Basic, or a macro language to drive an OLE Automation server regardless of the programming language used to write it.

OLE Automation is quite straightforward to implement in Delphi, thanks to the extensive work by the compiler and VCL to shield developers from its intricacies. To support OLE Automation, Delphi provides a wizard and a powerful type-library editor, and it supports dual interfaces.

When you use an in-process DLL, the client application can use the server and call its methods directly, because they are in the same address space. When you use OLE Automation, the situation is more complex. The client (called the *controller*) and the server are two separate applications running in different address spaces. For this reason, the system must dispatch the method calls using a complex mechanism called *marshaling* (something I won't cover in detail). What is important to know is that there are two ways a controller can call the methods exposed by a server:

- It can ask for the execution of a method, passing its name in a string, in a way similar to the dynamic call to a DLL. This is what Delphi does when you use a variant to call the OLE Automation server. This technique is very easy to use, but it is rather slow and provides very little compiler type-checking.
- It can import the definition of a Delphi interface for the object on the server and call its methods in a more direct way (simply dispatching a number). This technique, based on interfaces, allows the compiler to check the types of the parameters and produces faster code, but it requires a little more effort from the programmer. Also, you end up binding your controller application to a specific version of the server. A variation of this technique involves the use of dispatch interfaces, based on the definition of the interfaces.

In the following examples, we'll use all these techniques and compare them a little further.

Introducing Type Libraries

The most important difference between the two approaches is that the second generally requires a *type library*, one of the foundations of OLE and COM. A type library is basically a collection of type information. This collection generally describes all of the elements (the objects, the interfaces, and other type information) made available by a server. The key difference between a type library and other descriptions of these elements (such as some C or Pascal code) is that a type library is language-independent. The type elements are defined by OLE as a subset of the standard elements of programming languages, and any development tool can use them. Why do we need this information?

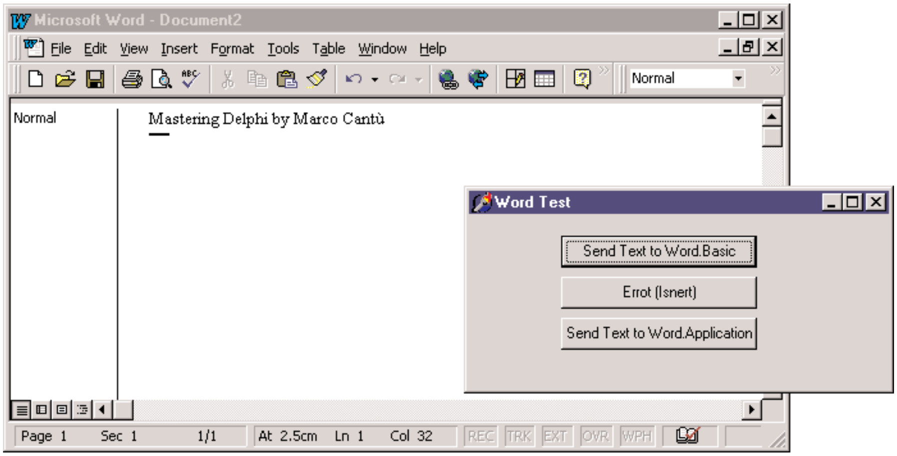
As mentioned before, an OLE Automation controller can use variants and have no type information about the server it is using. This means that, behind the scenes, every function call has to be dispatched to the server using the `Invoke` method of `IDispatch`, passing the function name as a string parameter and hoping the name corresponds to an existing function of the server.

Although this sounds difficult, a small code fragment using the old Automation interface of Microsoft Word, registered as `Word.Basic`, illustrates how simple it is for a programmer:

```
var
  VarW: Variant;
begin
  VarW := CreateOleObject ('Word.Basic');
  VarW.FileNew;
  VarW.Insert ('Mastering Delphi by Marco Cantù');
```

NOTE As we'll see later, recent versions of Word still register the `Word.Basic` interface, which corresponds to the internal WordBasic macro language, but it also registers the new interface `Word.Application`, which corresponds to the VBA macro language. We'll also see that Delphi provides some components that simplify the connection with Microsoft Office applications.

These three lines of code start Word (unless it was already running), create a new document, and add a few words to it. You can see the effect of this code in Figure 20.1. The code uses a variant, which is a *type-variant* data type. A variant can assume as its value different data types, including a COM object supporting the `IDispatch` interface. Variants are type-checked at run time; this is why the compiler can compile the code even if it doesn't know about the methods of the OLE Automation server.


Microsoft Word - Document2
File Edit View Insert Format Tools Table Window Help
Normal
Normal
Page 1 Sec 1 1/1 At 2.5cm Ln 1 Col 32 REC TRK EXT OVR WPH
Word Test
Send Text to Word.Basic
Error (Isnert)
Send Text to Word.Application

Mastering Delphi by Marco Cantù

FIGURE 20.1: This Word document is being created and composed by a Delphi application, WordTest.

Unfortunately, the Delphi compiler has no way to check whether the methods exist. Doing all the type checks at run time is risky, because if you make even a minor spelling error in a function name, you get no warning whatsoever of your error until you run the program and reach that line of code. For example, if you type `VarW.Isnert`, the compiler will not complain about the misspelling at all, but at run time, you'll get an error. Because it doesn't recognize the name, Word assumes the method does not exist.

Although the OLE `IDispatch` interface supports the approach we've just seen, it is also possible—and safer—for a server to export the description of its interfaces and objects using a type library. This type library can then be converted by a specific tool (such as Delphi) into definitions written in the language you want to use to write your client or controller program (such as Object Pascal). This makes it possible for a compiler to check whether the code is correct.

Once the compiler has done its checks, it can use either of two different techniques to send the request to the server. It can use a plain VTable (that is, an entry in an interface type declaration), or it can use a `dispinterface` (dispatch interface). We used an interface type declaration in the last chapter, so it should be familiar. A `dispinterface` is basically a way to map each entry in an interface to a number. Calls to the server can then be dispatched by number. We can consider this an intermediate technique, in between dispatching by function name and using a direct call in the VTable.

NOTE The term *dispinterface* is actually a keyword. A `dispinterface` is automatically generated by the type-library editor for every interface. Along with `dispinterface`, Delphi uses other related keywords: `dispid` indicates the number to associate with each element; `readonly` and `writeonly` are optional specifiers for properties.

The term used to describe this ability to connect to a server in two different ways, using a more dynamic or a more static approach, is *dual interfaces*. This means that in writing an OLE controller, you can choose to access the methods of a server in two ways: you can use late binding and the mechanism provided by the `dispinterface`, or you can use early binding and the mechanism based on the VTables, the interface types.

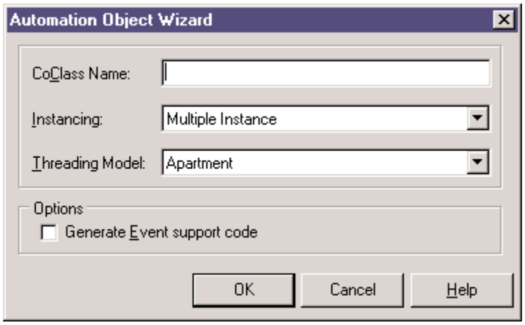
It is important to keep in mind that (along with other considerations) different techniques result in faster or slower execution. Looking up a function by name (and doing the type checking at run time) is the slowest approach, using a `dispinterface` is much faster, and using the direct VTable call is the fastest approach. We'll do this kind of test in the TlibCli example, later in this chapter.

Writing an OLE Automation Server

We'll start by writing an OLE Automation server. To create an OLE Automation object, you can use Delphi's Automation Object Wizard. Start with a new application, open the Object Repository by selecting File ➢ New, move to the ActiveX page, and choose Automation Object. In the resulting Automation Object Wizard (shown in Figure 20.2), enter the name of the class (without the initial *T*, because this will be added automatically for you) and click OK. Delphi will now open the type-library editor.

FIGURE 20.2:

Delphi's Automation Object Wizard


Automation Object Wizard

CoClass Name:

Instancing: Multiple Instance

Threading Model: Apartment

Options

□ Generate Event support code

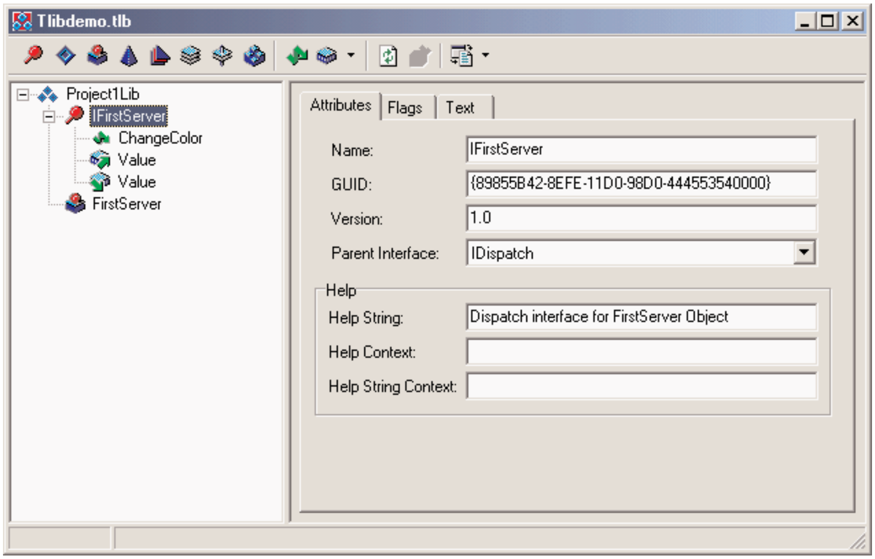
As you can see in Figure 20.2, Delphi can generate OLE Automation servers that also export events. Select the corresponding check box of the Wizard, and Delphi will add the proper entries in the type library and in the source code it generates.

The Type-Library Editor

The type-library editor is the tool you can use to define a type library in Delphi. Figure 20.3 shows its window after I've added some elements to it. The type-library editor allows you to add methods and properties to the OLE Automation server object we've just created. Once this is done, it can generate both the type library (TLB) file and the corresponding Object Pascal source code.

FIGURE 20.3:

The type-library editor, showing the details of an interface


Tlibdemo.tlb

- Project1Lib
  - IFirstServer
    - ChangeColor
    - Value
    - Value
  - FirstServer

Name: IFirstServer

GUID: {89855B42-8EFE-11D0-98D0-444553540000}

Version: 1.0

Parent Interface: IDispatch

Help

Help String: Dispatch interface for FirstServer Object

Help Context:

Help String Context:

To build a first example, we can add to the server a property and a method. In the editor, we actually add these two elements to the interface, which should be called `IFirstServer`. Select it, and then click the Method button of the toolbar. (The names of these buttons can be displayed by using the shortcut menu of the toolbar.) Now you have to give it a name, such as `ChangeColor`. You can type the name either in the Tree View control on the left side of the window or in the Name edit box on the right side. Delphi automatically defines the new method as a function in the Invoke Kind box and (as you'll see on the Parameters page) assigns it an `HRESULT` return value and no parameters. This corresponds to the Pascal definition:

```
procedure ChangeColor; safecall;
```

There are two reasons for this difference in the type of method. The first is that in the IDL language used by COM, all methods are indicated as functions (following the C language style); the second is that Delphi handles the `HRESULT` error codes automatically in every method that uses the `safecall` calling convention.

NOTE The methods contained in OLE Automation interfaces in Delphi generally use the `safecall` calling convention. This wraps a **`try`**/**`except`** block around each method and provides a default return value indicating error or success.

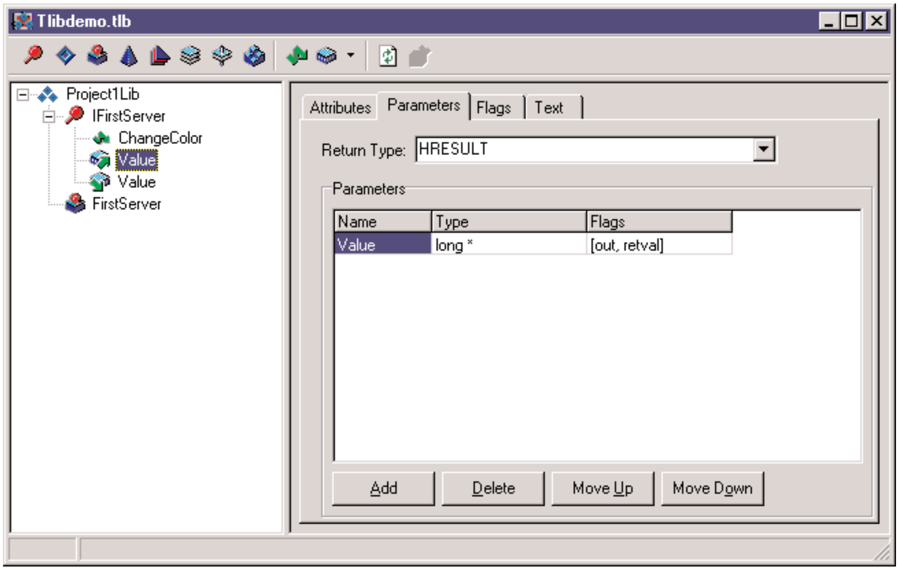
Now we can add a property to the interface by clicking the Property button of the type-library editor's toolbar. Again, we can type a name for it, such as `Value`, and select a data type in the Type combo box. Besides selecting one of the many types already listed, you can also enter other types directly, particularly interfaces of other objects. Keep in mind, however, that OLE Automation supports only a subset of Delphi types. In this example, we can select the `long` type, which corresponds to Delphi's `Integer` type.

If you look again in the Parameters page for this example (see Figure 20.4), you can see that both the `Set` and `Get` (actually called `Put` and `Get` in the COM jargon) methods have the `HRESULT` return value. You can also see that while the `Put` method uses the property's data type as its parameter (as with Delphi properties), the `Get` method uses a pointer to the type as its `out` parameter. This definition corresponds to the following elements of the Pascal interface:

```
function Get_Value: Integer; safecall;
procedure Set_Value(Value: Integer); safecall;
property Value: Integer read Get_Value write Set_Value;
```

FIGURE 20.4:

The Parameters page of the type-library editor



Return Type: HRESULT

Name	Type	Flags
Value	long *	[out, retval]

Clicking the Refresh button on the type-library editor toolbar generates the Pascal version of the interface. We'll examine it shortly, but first I want you to focus on the Text page of the editor, which includes the definition we've just created, written in the IDL language:

```
interface IFirstServer: IDispatch
{
  [id(0x00000001)]
  HRESULT _stdcall ChangeColor( void );
  [propget, id(0x00000002)]
  HRESULT _stdcall Value([out, retval] long * Value );
  [propput, id(0x00000002)]
  HRESULT _stdcall Value([in] long Value );
};
```

Fortunately, Delphi's type-library editor saves you from writing similar code by hand, and the Delphi environment options (in the Type Library page) include a radio button to select Pascal or IDL in the text displayed by the type-library editor.

The Code of the Server

Now we can close the type-library editor and save the changes. This operation adds three items to the project: the type library file, a corresponding Pascal definition, and the declaration of the server object. The type library is connected to the project using a resource-inclusion statement, added to the source code of the project file:

```
{$R *.TLB}
```

You can always reopen the type-library editor by using the View ➢ Type Library command or by selecting the proper TLB file in the normal File Open dialog box of Delphi.

As mentioned earlier, the type library is also converted into an interface definition and added to a new Pascal unit. This unit is quite long, so I've listed in the book only its key elements. The most important part is the new interface declaration:

```
type
  IFirstServer = interface(IDispatch)
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; safecall;
    function Get_Value: Integer; safecall;
    procedure Set_Value(Value: Integer); safecall;
    property Value: Integer read Get_Value write Set_Value;
  end;
```

Then comes the `dispinterface`, which associates a number with each element of the `IFirstServer` interface:

```
type
  IFirstServerDisp = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
  end;
```

The last portion of the file includes the so-called CoClass (also shown in the type-library editor), a class used to create an object on the server (and for this reason used on the client side of the application, not on the server side):

```
type
  CoFirstServer = class
    class function Create: IFirstServer;
    class function CreateRemote(const MachineName: string): IFirstServer;
  end;
```

All the declarations of this file (I've skipped some others) can be considered an internal, hidden implementation support. You don't need to understand them fully in order to write most OLE Automation applications.

Finally, Delphi generates a file with the declaration of the actual object. This unit is added to the application and is the one we'll work on to finish the program. This unit declares the class of the server object, which must implement the interface we've just defined:

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
  end;
```

Delphi already provides us with the skeleton code of the methods, so you only need to fill the lines in between. This is the final code of the server object methods of the TLibDemo example from the companion CD:

```
function TFirstServer.Get_Value: Integer;
begin
  Result := ServerForm.Value;
end;

procedure TFirstServer.ChangeColor;
begin
  ServerForm.ChangeColor;
end;

procedure TFirstServer.Set_Value(Value: Integer);
begin
  ServerForm.Value := Value;
end;
```

In this case, the three methods refer to a property and two methods I've added to the form. In general, you should not add code related to the user interface inside the class of the server object. It is better to refer to a user interface element, such as a form class, and let it perform the actions.

I've added a property to the form because I want to change the `Value` property and have a side effect (displaying the value in an edit box). The server object, in this example, exposes some properties and methods of the application. Here is the part of the declaration of the `TServerForm` class I've edited manually:

```
type
  TServerForm = class(TForm)
    ...
  private
    CurrentValue: Integer;
  protected
    procedure SetValue (NewValue: Integer);
  public
    property Value: Integer read CurrentValue write SetValue;
    procedure ChangeColor;
  end;
```

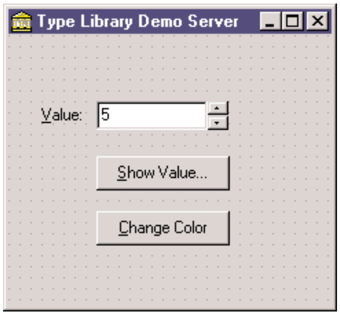
The implementation of these methods is quite straightforward, and you can easily guess what their code looks like. What's important is the `SetValue` method, which might produce a side effect:

```
procedure TServerForm.SetValue (NewValue: Integer);
begin
  if NewValue <> CurrentValue then
  begin
    CurrentValue := NewValue;
    UpDown1.Position := CurrentValue;
  end;
end;
```

The form of this example has an edit box with an associated UpDown component as well as a couple of buttons to show the current value and change the color. You can see this form at design time in Figure 20.5.

FIGURE 20.5:

The form of the TLibDemo example at design time



Registering the Automation Server

The unit containing the server object has one more statement, added by Delphi to the `initialization` section:

```
initialization
  TAutoObjectFactory.Create(ComServer, TFirstServer, Class_FirstServer,
    ciMultiInstance);
end.
```

NOTE In this case, I've selected multiple instancing. For the various instancing styles possible in COM, see the sidebar "COM Instancing and Threading Models" in Chapter 19, "COM Programming."

This is not very different from the creation of class factories we saw in the examples of the last chapter. Combined with the call to the `Initialize` method of the `Application` object, which Delphi adds by default to the project source code of any program, the `initialization` code above makes the registration of this server straightforward.

You can add the server information to the Windows Registry by running this application on the target machine (the computer where you want to install the OLE Automation server), passing to it the `/regserver` parameter on the command line. You can do this by selecting Start ➢ Run, by using the Explorer or File Manager, or by running the program within

Delphi after you've entered a command-line parameter (using the Run ➢ Parameters command). Another command-line parameter, `/unregserver`, is used to remove this server from the Registry.

Writing a Client for Our Server

Now that we have built a server, we can prepare a client program to test it. This client can connect to the server either by using variants or by using the new type library. This second approach can be implemented manually or by using the techniques introduced in Delphi 5 for wrapping components around Automation servers. We'll actually try out all of these approaches.

Create a new application—I've called it TLibCli—and then open the type library file of the server, after (optionally) copying it to the project's directory. Save the type library file, using Delphi's File ➢ Save menu command, and a new version of the interface declarations will be generated for you. Of course, in this case you could have grabbed the Pascal declarations from the server source code, but I'm trying to follow a more general approach, which also applies when you haven't written the server yet. In fact, you can usually extract the type library directly from the executable file of the server or from a DLL shipped with the program.

WARNING Do not add the type library to the client application, though, because we are writing the OLE Automation controller, not a server. The Delphi project of a controller should not include the type library of the server it connects to.

You can refer to the Pascal file generated by the type-library editor in the code of the main form:

```
uses
  TlibdemoLib_TLB;
```

I've already mentioned that one of the elements of this unit generated by the type library is the *creation* class, or CoClass, a special class with two class functions you can use to create a server object locally or remotely (using DCOM). I've already shown you the interface of this class, but here is the implementation:

```
class function CoFirstServer.Create: IFirstServer;
begin
  Result := CreateComObject(Class_FirstServer) as IFirstServer;
end;

class function CoFirstServer.CreateRemote(
  const MachineName: string): IFirstServer;
begin
  Result := CreateRemoteComObject(MachineName, Class_FirstServer)
    as IFirstServer;
end;
```

You can use the first of these two functions, `Create`, to create a server object (and possibly start the server application) on the same computer. You can use the second function, `CreateRemote`, to create the server on a different computer, as long as your version of the operating system supports DCOM.

The two functions are a shortcut of the `CreateComObject` call, which allows you to create an instance of a COM object if you know its GUID. As an alternative, you can also use the `CreateOleObject` function, which requires as a parameter the registered name of the server. There is another difference between these two creation functions: `CreateComObject` returns an object of the `IUnknown` type, whereas `CreateOleObject` returns an object of the `IDispatch` type.

In my example, I'm going to use the `CoFirstServer.Create` shorthand. When you create the server object, you get as return value an `IFirstServer` interface. You can use it directly or store it in a variant variable. Here is an example of the first approach:

```
var
  MyServer: Variant;
begin
  MyServer := CoFirstServer.Create;
  MyServer.ChangeColor;
```

This code, based on variants, is not very different from that of the first controller we built in this chapter (the one that used Microsoft Word). Here is the alternative code, which has exactly the same effect:

```
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

Interfaces, Variants, and Dispatch Interfaces: Testing the Speed Difference

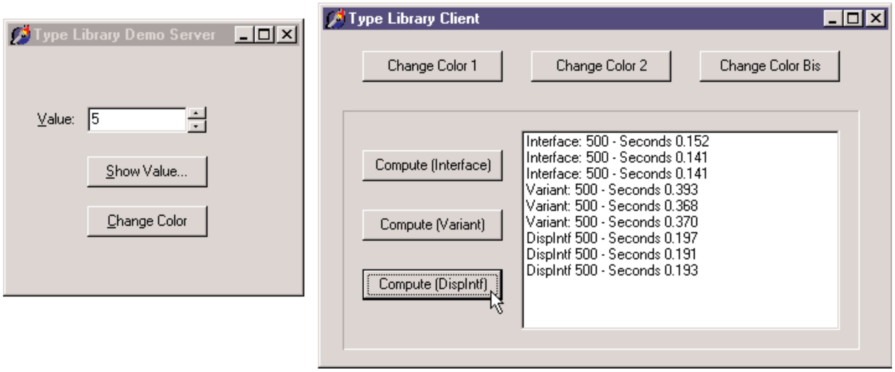
As I mentioned in the section introducing type libraries, one of the differences between these approaches is speed. It is actually quite complex to assess the exact performance of each technique because there are many factors involved. I've added a simple test to the TLibCli example on the companion CD, just to give you an idea. Here is the code of the test, a loop that accesses the `Value` of the server. The total value is displayed only to fool the optimizer, which might otherwise remove some of the code. The real output of the program relates to the timing, which is determined by calling the `GetTickCount` API function before and after executing the loop. (Two alternatives are to use Delphi's own time functions, which are slightly less precise, or to use the very precise timing functions of the multimedia support unit, MMSystem.) Here is the code of one of the methods; they are quite similar:

```
procedure TClientForm.BtnIntfClick(Sender: TObject);
var
  I, K: Integer;
  Ticks: Cardinal;
begin
  Screen.Cursor := crHourglass;
  try
    Ticks := GetTickCount;
    K := 0;
    for I := 1 to 100 do
      K := K + IMyServer.Value;
    Ticks := GetTickCount - Ticks;
    ListResult.items.Add (Format (
      'Interface: %d - Seconds %.3f', [K, Ticks / 1000]));
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

With this program, you can compare the output obtained by calling this method based on an interface, the corresponding version based on a variant, and even a third version based on a dispatch interface. An example of the output (which is added to a list box so you can do several tests and compare the results) is shown in Figure 20.6. Obviously, the timing depends on the speed of your computer, and you can also alter the results by increasing or decreasing the maximum value of the loop counter.

FIGURE 20.6:

The TLibCli OLE Automation controller can access the server in different ways, with different performance results. Notice the server window in the background.



Value: 5

```
Interface: 500 - Seconds 0.152
Interface: 500 - Seconds 0.141
Interface: 500 - Seconds 0.141
Variant: 500 - Seconds 0.393
Variant: 500 - Seconds 0.368
Variant: 500 - Seconds 0.370
DispIntf 500 - Seconds 0.197
DispIntf 500 - Seconds 0.191
DispIntf 500 - Seconds 0.193
```

We've already seen how you can use the interface and the variant. What about the dispatch interface? You can declare a variable of the dispatch interface type, in this case:

```
var
  DMyServer: IFirstServerDisp;
```

Then you can use it to call the methods as usual, after you've assigned an object to it by casting the object returned by the CoClass:

```
DMyServer := CoFirstServer.Create as IFirstServerDisp;
```

Looking at the timing and at the internal code of the example, there is apparently very little difference between the use of the interface and of the dispatch interface, because the two are actually connected. In other words, we can say that dispatch interfaces are a technique in between variants and interfaces, but they deliver almost all of the speed of interfaces.

The Scope of Automation Objects

Another important element to keep in mind is the *scope* of the automation objects. Variants and interface objects use reference-counting techniques, so if a variable that is related to an interface object is declared locally in a method, at the end of the method the object will be destroyed and the server may terminate (if all the objects created by the server have been destroyed). For example, writing a method with this code produces very little effect:

```
procedure TClientForm.ChangeColor;
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
end;
```

Unless the server is already active, a copy of the program is created, and the color is changed, but then the server is immediately closed as the interface-typed object goes out of scope. The alternative approach I've used in the TLibCli example is to declare the object as a field of the form and create the COM objects at start-up, as in this procedure:

```
procedure TClientForm.FormCreate(Sender: TObject);
begin
  IMyServer := CoFirstServer.Create;
end;
```

With this code, as the client program starts, the server program is immediately activated. At the program termination, the form field is destroyed and the server closes. A further alternative is to declare the object in the form, but then create it only when it is used, as in these two code fragments:

```
// MyServerBis: Variant;
if varType (MyServerBis) = varEmpty then
  MyServerBis := CoFirstServer.Create;
MyServerBis.ChangeColor;

// IMyServerBis: IFirstServer;
if not Assigned (IMyServerBis) then
  IMyServerBis := CoFirstServer.Create;
IMyServerBis.ChangeColor;
```

NOTE A variant is initialized to the `varEmpty` type when it is created. If you instead assign the value null to the variant, its type becomes `varNull`. Both `varEmpty` and `varNull` represent variants with no value assigned, but they behave differently in expression evaluation. The `varNull` value always propagates through an expression (making it a null expression), while the `varEmpty` value quietly disappears.

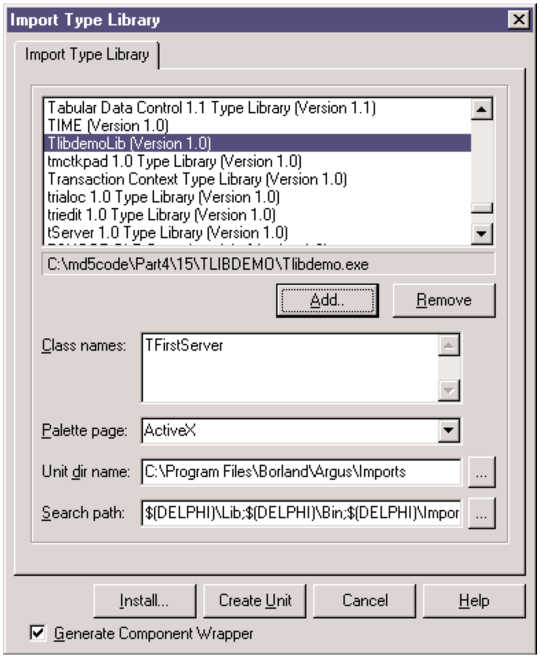
The Server in a Component

When creating a client program for our server or any other Automation server, we can use a better approach, namely, wrapping a Delphi component around the COM server. Actually, if you look at the final portion of the `TlibdemoLib_TLB` file, you can find the following declaration:

```
// OLE Server Proxy class declaration
TFirstServer = class(TOleServer)
private
  FIntf: IFirstServer;
  FProps: TFirstServerProperties;
  function GetServerProperties: TFirstServerProperties;
  function GetDefaultInterface: IFirstServer;
protected
  procedure InitServerData; override;
  function Get_Value: Integer;
  procedure Set_Value(Value: Integer);
public
  constructor Create(AOwner: TComponent); override;
  destructor  Destroy; override;
  procedure Connect; override;
  procedure ConnectTo(svrIntf: IFirstServer);
  procedure Disconnect; override;
  procedure ChangeColor;
  property  DefaultInterface: IFirstServer read GetDefaultInterface;
  property Value: Integer read Get_Value write Set_Value;
published
  property Server: TFirstServerProperties read GetServerProperties;
end;
```

This is a new component, derived from `TOleServer`, that the system registers in the `Register` procedure, which is part of the unit. If you add this unit to a package, the new server component will become available on the Delphi Component Palette. You can also import the type library of the new server (with the Project ➢ Import Type Library menu command), add the server to the list (by clicking the Add button and selecting the server's executable file), and install it in a new or existing package. The component will be placed in the Servers page of the Palette. The Import Type Library dialog box indicating these operations is visible in Figure 20.7.

FIGURE 20.7: The Import Type Library dialog box can be used to import an Automation server object as a new Delphi component.



Tabular Data Control 1.1 Type Library (Version 1.1)
TIME (Version 1.0)
TlibdemoLib (Version 1.0)
tmctkpad 1.0 Type Library (Version 1.0)
Transaction Context Type Library (Version 1.0)
trialoc 1.0 Type Library (Version 1.0)
triedit 1.0 Type Library (Version 1.0)
tServer 1.0 Type Library (Version 1.0)

C:\md5code\Part4\15\TLIBDEMO\Tlibdemo.exe

Class names: TFirstServer

Palette page: ActiveX

Unit dir name: C:\Program Files\Borland\Argus\Imports

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Impor

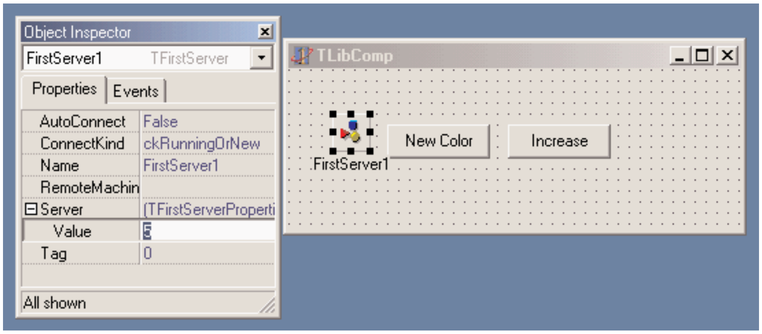
☑ Generate Component Wrapper

I've created a new package, PackAuto, available in the directory of the TlibDemo project. In this package, I've added the directive `LIVE_SERVER_AT_DESIGN_TIME` in the Directories/Conditionals page of the Project Options dialog box of the package. This enables an extra feature that you don't get by default: at design time, the server component will have an extra property that lists as subitems all the properties of the Automation server. You can see an example in Figure 20.8, taken from the TLibComp example at design time.

WARNING The `LIVE_SERVER_AT_DESIGN_TIME` directive should be used with care with the most complex Automation servers (including programs such as Word, Excel, PowerPoint, and Visio). In fact, this setting requires the application to be in a particular mode before you can use some properties of their automation interfaces. For example, you'll get exceptions if you touch the Word server before a document has been opened in Word. That's why this feature is not active by default in Delphi—it's problematic at design time for many servers.

FIGURE 20.8:

A server component, with the live properties at design time



Object Inspector

FirstServer1 TFirstServer

Properties | Events

Property	Value
AutoConnect	False
ConnectKind	ckRunningOrNew
Name	FirstServer1
RemoteMachin	
⊟Server	(TFirstServerProperti
Value	5
Tag	0

All shown

As you can see in the Object Inspector, the component has few properties. AutoConnection indicates when to start up the server component at design time and as soon as the client program starts. As an alternative, the Automation server is started the first time one of its methods is called. Another property, ConnectKind, indicates how to establish the connection with the server. It can always start a new instance (ckNewInstance), use the running instance (ckRunningInstance, which causes an access violation if the server is not already running), or select the current instance or start a new one if none is available (ckRunningOrNew). Finally, you can ask for a remote server with ckRemote and directly attach a server in the code after a manual connection with ckAttachToInterface.

OLE Data Types

OLE and COM do not support all of the data types available in Delphi. This is particularly important for OLE Automation, because the client and the server are often executed in different address spaces, and the system must move the data from one side to the other. Also keep in mind that OLE interfaces should be accessible by programs written in any language.

COM data types include basic data types such as Integer, SmallInt, Byte, Single, Double, WideString, Variant, and WordBool (but not Boolean). Table 20.1 presents the mapping of some basic data types, available in the type-library editor, to the corresponding Delphi types.

TABLE 20.1: OLE and Delphi Data Types

OLE Type	Delphi Type
BSTR	WideString
byte	ShortInt
CURRENCY	Currency
DATE	TDateTime
DECIMAL	TDecimal

Continued on next page

TABLE 20.1 continued: OLE and Delphi Data Types

OLE Type	Delphi Type
double	Double
float	Single
GUID	GUID
int	SYSINT
long	Integer
LPSTR	PChar
LPWSTR	PWideChar
short	SmallInt
unsigned char	Byte
unsigned int	SYSUINT
unsigned long	UINT
unsigned short	Word
VARIANT	OleVariant

Notice that SYSINT is currently defined as an Integer, so don't worry about the apparently strange type definition. Besides the basic data types, you can also use OLE types for complex elements such as fonts, string lists, and bitmaps, using the IFontDisp, IStrings, and IPictureDisp interfaces. The following sections describe the details of a server that provides a list of strings and a font to a client.

Exposing Strings Lists and Fonts

The ListServ example is a practical demonstration of how you can expose two complex types, such as a list of strings and a font, from an OLE Automation server written in Delphi. I've chosen these two specific types simply because they are both supported by Delphi.

The IFontDisp interface is actually provided by Windows and is available in the ActiveX unit. The AxCtrls Delphi unit extends this support by providing conversion methods like GetOleFont and SetOleFont. The IStrings interface is provided by Delphi in the StdVCL unit, and the AxCtrls unit provides conversion functions for this type (along with a third type I'm not going to use, TPicture).

WARNING To run this and similar applications, the StdVCL library must be installed and registered on the client computer. On your computer, it is registered during Delphi's installation.

The server we are building has a plain form containing a list-box component. It includes an Automation object built around the following interface:

```
type
  IListServer = interface (IDispatch)
    ['{323C4A84-E400-11D1-B9F1-004845400FAA}']
    function Get_Items: IStrings; safecall;
    procedure Set_Items(const Value: IStrings); safecall;
    function Get_Font: IFontDisp; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    property Items: IStrings read Get_Items write Set_Items;
    property Font: IFontDisp read Get_Font write Set_Font;
  end;
```

The server object has the same four methods listed in its interface as well as some private data storing the status, the initialization function, and the destructor:

```
type
  TListServer = class (TAutoObject, IListServer)
  private
    fItems: TStrings;
    fFont: TFont;
  protected
    function Get_Font: IFontDisp; safecall;
    function Get_Items: IStrings; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Items(const Value: IStrings); safecall;
  public
    destructor Destroy; override;
    procedure Initialize; override;
  end;
```

The code of the methods is limited to few statements. The pseudoconstructor creates the internal objects, and the destructor destroys them. Here is the first of the two:

```
procedure TListServer.Initialize;
begin
  inherited Initialize;
  fItems := TStringList.Create;
  fFont := TFont.Create;
end;
```

The `Set` and `Get` methods copy information from the OLE interfaces to the local data and then from this to the form and vice versa. The two methods of the strings, for example, do this by calling the `GetOleStrings` and `SetOleStrings` Delphi functions.

After we've compiled and registered the server, we can turn our attention to the client application. This embeds the Pascal translation of the type library of the server, as in the previous example, and then implements an object that uses the interface. Instead of creating the server when the object starts, the client program creates it when it is required. I've described this technique earlier, but the problem is that because there are several buttons a user can click, and we don't want to impose an order, every event should have a handler like this:

```
if not Assigned (ListServ) then
  ListServ := CoListServer.Create;
```

This kind of code duplication is quite dangerous, so I've decided to use an alternative approach. I've defined a property corresponding to the interface of the server and defined a read method for it. The property is mapped to some internal data I've defined with a different name to avoid the error of using it directly. Here are the definitions added to the form class:

```
private
  fInternalListServ: IListServer;
  function GetListSrv: IListServer;
public
  property ListSrv: IListServer read GetListSrv;
```

The implementation of the `Get` method can check whether the object already exists. This code is going to be repeated often, but that should not slow down the application noticeably:

```
function TListCliForm.GetListSrv: IListServer;
begin
  // eventually create the server
  if not Assigned (fInternalListServ) then
    fInternalListServ := CoListServer.Create;
  Result := fInternalListServ;
end;
```

You can see an example of the client application running (along with the server) in Figure 20.9.

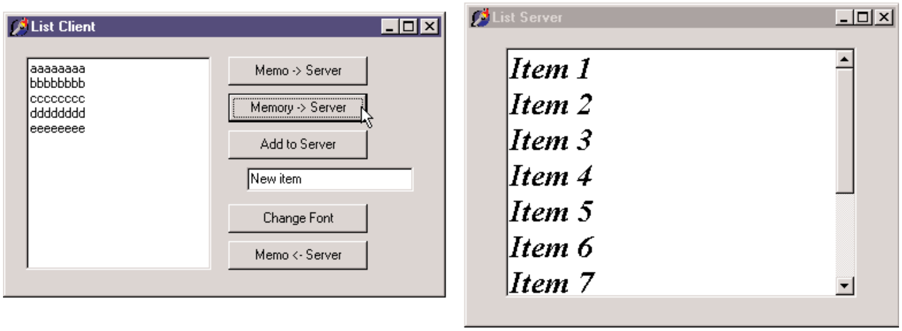


FIGURE 20.9: The ListCli and ListServ applications share complex data, namely fonts and lists of strings.

This is an example of the selection of a font, which is then sent to the server:

```
procedure TListCliForm.btnFontClick(Sender: TObject);
var
  NewFont: IFontDisp;
```

```
begin
  // select a font and apply it
  if FontDialog1.Execute then
  begin
    GetOleFont (FontDialog1.Font, NewFont);
    ListSrv.Font := NewFont;
  end;
end;
```

There are also several methods related to the strings, which you can see by looking at the source code of the program.

Using Office Programs

So far, we've built both the client and the server side of the OLE Automation connection. If your aim is just to let two applications you've built cooperate, this is certainly a useful technique, although it is not the only one. We've seen some alternative data-sharing approaches in the last two chapters (using memory-mapped files and the `wm_CopyData` message). The real value of OLE Automation is that it is a standard, so you can use it to integrate your Delphi programs with other applications your users own. A typical example is the integration of a program with office applications, such as Microsoft Word and Microsoft Excel, or even with stand-alone applications, such as AutoCAD.

Integration with these applications provides a two-fold advantage:

- You can let your users work in an environment they know—for example, generating reports and memos from database data in a format they can easily manipulate.
- You can avoid implementing complex functionality from scratch, such as writing your own word-processing code inside a program. Instead of just reusing components, you can reuse complex applications.

There are also some drawbacks with this approach, which are certainly worth mentioning:

- The user must own the application you plan to integrate with, and they may also need a recent version of it to support all the features you are using in your program.
- You have to learn a new programming language and programming structure, often with limited documentation at hand. It is true, of course, that you are still using Pascal, but the code you write depends on the OLE data types, the types introduced by the server, and in particular, a collection of interrelated classes that are often difficult to understand.
- You might end up with a program that works only with a specific version of the server application, particularly if you try to optimize the calls by using interfaces instead of variants. In particular, Microsoft does not attempt to maintain script compatibility between major releases of Word or other Office applications.

We've already seen a small source code excerpt from the WordTest example, but now I want to complete the coverage of this limited but interesting test program by providing a few extra features.

Sending Data to Microsoft Word

Delphi simplifies the use of Microsoft Office applications by preinstalling some ready-to-use components that wrap the Automation interface of these servers. These components, available in the Servers page of the Palette, have been installed using the same technique I demonstrated in the last section.

NOTE What I want to underline here is that the real plus of Delphi lies in this technique of creating components to wrap existing Automation servers, rather than in the availability of some predefined server components.

Technically, it is possible to use variants to interact with Automation servers, as we've seen in the section "Introducing Type Libraries." Using interfaces and the type libraries is certainly better, because the compiler helps you catch errors in the source code and produces faster code. Thanks to the new server component, this process is also quite straightforward.

I've written a program, called DBOffice, which uses predefined server components to send a table to Word and to Excel. In both cases, you can use the application object, the document/worksheet object, or a combination of the two. There are other specialized components, for tasks such as handling Excel charts, but this example will suffice to introduce use of the built-in Office components.

NOTE The DBOffice program was tested with Office 97. I'm currently using StarOffice more often than the Microsoft suite, so I never feel compelled to give Microsoft more money by upgrading to their newer offerings.

In case of Microsoft Word, I use only a document object with default settings. The code used to send the table to Word starts by adding some text to a document:

```
procedure TFormOff.BtnWordClick(Sender: TObject);
begin
  WordDocument1.Activate;
  // insert title
  WordDocument1.Range.Text := 'American Capitals from ' + Table1.TableName;
  WordDocument1.Range.Font.Size := 14;
```

This code follows the typical `while` loop, which scans the database table and has the following code inside:

```
while not Table1.EOF do
begin
  // send the two fields
```

```
    WordDocument1.Range.InsertParagraphAfter;
    WordDocument1.Paragraphs.Last.Range.Text :=
      Table1.FieldByName ('Name').AsString + #9 +
      Table1.FieldByName ('Capital').AsString;
    Table1.Next;
  end;
```

The final part of the code gets a little more complex. It works on a selection and on a row of the table, respectively stored in two variables of the `Range` and `Row` types defined by Word and available in the Word97 unit (the program will have to be updated if you choose the Office 2000 version of the server component while installing Delphi).

```
procedure TFormOff.BtnWordClick(Sender: TObject);
var
  RangeW: Word97.Range;
  v1: Variant;
  ov1: OleVariant;
  Row1: Word97.Row;
begin
  // code above...
  RangeW := WordDocument1.Content;
  v1 := RangeW;
  v1.ConvertToTable (#9, 19, 2);
  Row1 := WordDocument1.Tables.Item(1).Rows.Get_First;
  Row1.Range.Bold := 1;
  Row1.Range.Font.Size := 30;
  Row1.Range.InsertParagraphAfter;
  ov1 := ' ';
  Row1.ConvertToText (ov1);
end;
```

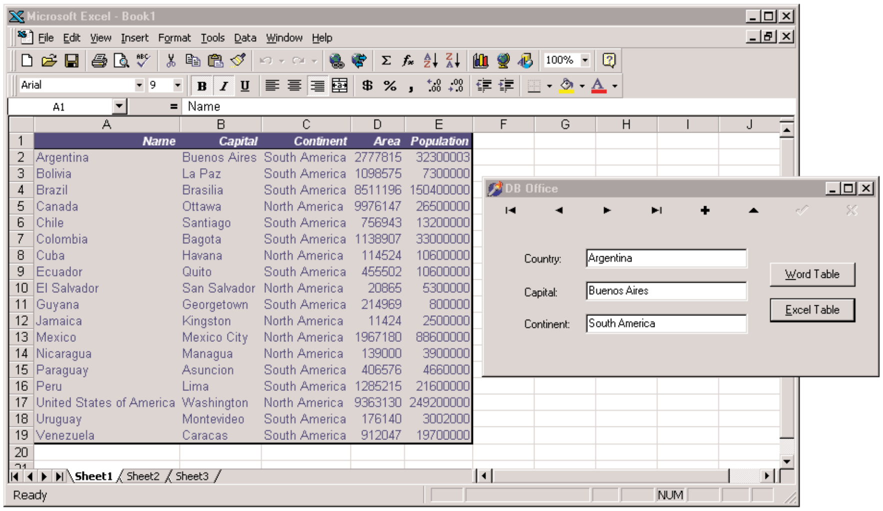
As you can see in the last statement above, in order to pass a parameter, you must first save it in an `OleVariant` variable, because many parameters are passed by reference, so you cannot pass a constant value. This implies that if there are many parameters, you must still define some, even if you are fine with the default values. An often-useful alternative is to use a temporarily variant variable and apply the method to it, because variants don't require strict type-checking on the parameters. This technique is used in the code above to call the `ConvertToTable` method, which has more than 10 parameters.

Building an Excel Table

In the case of Excel, I've used a slightly different approach and worked with the application object. The code creates a new Excel spreadsheet, fills it with a database table, and formats the result. It uses an Excel internal object, Range, which is not to be confused with a similar type available in Word (the reason this type is prefixed with the name of the unit defining the Excel type library). Here is the complete code:

```
procedure TFormOff.BtnExcelClick(Sender: TObject);
var
  RangeE: Excel97.Range;
  I, Row: Integer;
  Bookmark: TBookmarkStr;
begin
  // create and show
  ExcelApplication1.Visible [0] := True;
  ExcelApplication1.Workbooks.Add (NULL, 0);
  // fill is the first row with field titles
  RangeE := ExcelApplication1.ActiveCell;
  for I := 0 to Table1.Fields.Count - 1 do
  begin
    RangeE.Value := Table1.Fields [I].DisplayLabel;
    RangeE := RangeE.Next;
  end;
  // add field data in following rows
  Table1.DisableControls;
  try
    Bookmark := Table1.Bookmark;
    try
      Table1.First;
      Row := 2;
      while not Table1.EOF do
      begin
        RangeE := ExcelApplication1.Range ['A' + IntToStr (Row),
          'A' + IntToStr (Row)];
        for I := 0 to Table1.Fields.Count - 1 do
        begin
          RangeE.Value := Table1.Fields [I].AsString;
          RangeE := RangeE.Next;
        end;
        Table1.Next;
        Inc (Row);
      end;
    finally
      Table1.Bookmark := Bookmark;
    end;
  finally
    Table1.EnableControls;
  end;
  // format the section
  RangeE := ExcelApplication1.Range ['A1', 'E' + IntToStr (Row - 1)];
  RangeE.AutoFormat (3, NULL, NULL, NULL, NULL, NULL, NULL);
end;
```

You can see the effect of this code in Figure 20.10. Notice that in the code I don't handle any events of the Office applications, but many are available. Handling these events was quite complex in the past, but they now become as simple to handle as events of native Delphi components. The presence of these events is a reason to have specific objects for documents and other specific elements: you might want to know when the user closes a document, and that therefore this is an event of the document object, not of the application object.



	A	B	C	D	E
1	Name	Capital	Continent	Area	Population
2	Argentina	Buenos Aires	South America	2777815	32300003
3	Bolivia	La Paz	South America	1098575	7300000
4	Brazil	Brasilia	South America	8511196	150400000
5	Canada	Ottawa	North America	9976147	26500000
6	Chile	Santiago	South America	756943	13200000
7	Colombia	Bagota	South America	1138907	33000000
8	Cuba	Havana	North America	114524	10600000
9	Ecuador	Quito	South America	455502	10600000
10	El Salvador	San Salvador	North America	20865	5300000
11	Guyana	Georgetown	South America	214969	800000
12	Jamaica	Kingston	North America	11424	2500000
13	Mexico	Mexico City	North America	1967180	88600000
14	Nicaragua	Managua	North America	139000	3900000
15	Paraguay	Asuncion	South America	406576	4660000
16	Peru	Lima	South America	1285215	21600000
17	United States of America	Washington	North America	9363130	249200000
18	Uruguay	Montevideo	South America	176140	3002000
19	Venezuela	Caracas	South America	912047	19700000

Country: Argentina

Capital: Buenos Aires

Continent: South America

FIGURE 20.10:

The Excel spreadsheet \generated by the DBOffice application

NOTE When using the Office server components, one of the key problems is the lack of adequate documentation. Although Microsoft distributes some of it with the high-end version of the Office suite, this is certainly not Delphi friendly. A totally alternative approach to solve the problem is to use OfficePartner, a set of components from TurboPower Software (**`www.turbopower.com`**). These components map the Office servers, like those available in Delphi, but they also provide extensive property editors that allow you to work visually with the internal structure of these servers. With these property editors, you can create documents, paragraphs, tables, and all the other internal objects even at design time! From my experience, this can really save a lot of time.

Using Compound Documents

Compound documents, or active documents, are Microsoft's names for the technology that allows in-place editing of a document within another one (for example, a picture in a Word document). This is the technology that originated the term OLE, but although it is still in use, its role is definitely more limited than Microsoft envisioned when it was introduced in the early 1990s. Compound documents actually have two different capabilities, o*bject linking* and *embedding* (hence the term OLE):

- Embedding an object in a compound document corresponds to a smart version of the copy and paste operations you make with the Clipboard. The key difference is that when you copy an OLE object from a server application and paste it into a container application, you copy both the data and some information about the server (its GUID). This allows you to activate the server application from within the container to edit the data.
- Linking an object to a compound document instead copies only a reference to the data and the information about the server. You generally activate object linking by using the Clipboard and making a Paste Link operation. When editing the data in the container application, you'll actually modify the original data, which is stored in a separate file.

Because the server program refers to an entire file (only part of which might be linked in the client document), the server will be activated in a stand-alone window, and it will act upon the entire original file, not just the data you've copied. When you have an embedded object, instead, the container might support visual (or *in-place*) editing, which means that you can modify the object in context, inside the container's main window. The server and container application windows, their menus, and their toolbars are merged automatically, allowing the user to work within a single window on several different object types—and therefore with several different OLE servers—without leaving the window of the container application.

Another key difference between embedding and linking is that the data of an embedded object is stored and managed by the container application. The container saves the embedded object in its own files. By contrast, a linked object physically resides in a separate file, which is handled by the server exclusively, even if the link refers only to a small portion of the file.

In both cases, the container application doesn't have to know how to handle the object and its data—not even how to display it—without the help of the server. Accordingly, the server application has a lot of work to do, even when you are not editing the data. Container applications often make a copy of the image of an OLE object and use the bitmap to represent the data, which speeds up some operations with the object itself. The drawback of this approach is that many commercial OLE applications end up with bloated files (because two copies of the same data are saved). If you consider this problem along with the relative slowness of OLE and the amount of work necessary to develop OLE servers, you can understand why the use of this powerful approach is still somewhat limited, compared with what Microsoft envisioned a few years ago.

Compound document containers can support OLE in varying degrees. You can place an object in a container by inserting a new object, by pasting or *paste-linking* one from the Clipboard, by dragging one from another application, and so on.

Once the object is placed inside the container, you can then perform operations on it, using the server's available *verbs*, or actions. Usually the *edit verb* is the default action—the action performed when you double-click on the object. For other objects, such as video or sound clips, *play* is defined as the default action. You can typically see the list of actions supported by the current contained object by right-clicking it. The same information is available in many programs via the Edit ➢ Object menu item, which has a submenu that lists the available verbs for the current object.

NOTE Delphi provides no *visual* support for building compound document servers. You can always write a server implementing the proper interfaces. Compound document container support, instead, is easily available through the OleContainer component.

The OLE Container Component

To create an OLE container application in Delphi, place an OleContainer component in a form. Then select the component and right-click to activate its shortcut menu, which will have an Insert Object command. When you select this command, Delphi displays the standard OLE Insert Object dialog box. This dialog box allows you to choose from one of the server applications registered on the computer.

Once the OLE object is inserted in the container, the shortcut menu of the control container component will have several more custom menu items. The new menu items include commands to change the properties of the OLE object, insert another one, copy the existing object, or remove it. The list also includes the verbs, or actions, of the object (such as Edit, Open, or Play). Once you have inserted an OLE object in the container, the corresponding server will launch to let you edit the new object. As soon as you close the server application, Delphi updates the object in the container and displays it at design time in the form of the Delphi application you are developing.

If you look at the textual description of a form containing a component with an object inside, you'll notice a `Data` property, which contains the actual data of the OLE object. Although the client program stores the data of the object, it doesn't know how to handle and show that without the help of the proper server (which must be available on the computer where you run the program). This means that the OLE object is *embedded*.

To fully support compound documents, a program should provide a menu and a toolbar or panel. These extra components are important because in-place editing implies a merging of the user interface of the client and that of the server program. When the OLE object is activated in place, some of the pull-down menus of the server application's menu bar are added to the menu bar of the container application.

OLE menu merging is handled almost automatically by Delphi. You only need to set the proper indexes for the menu items of the container, using the GroupIndex property. Any menu item with an odd index number is replaced by the corresponding element of the active OLE object. More specifically, the File (0) and Window (4) pull-down menus belong to the container application. The Edit (1), View (3), and Help (5) pull-down menus (or the groups of pull-down menus with those indexes) are taken by the OLE server. A sixth group, named Object and indicated with the index 2, can be used by the container to display another pull-down menu between the Edit and View groups, even when the OLE object is active. The OleCont demo program I've written to demonstrate these features allows a user to create a new object by calling the InsertObjectDialog method of the TOleContainer class.

The InsertObjectDialog method shows a system dialog box, but it doesn't automatically activate the OLE object:

```
procedure TForm1.New1Click(Sender: TObject);
begin
  if OleContainer1.InsertObjectDialog then
    OleContainer1.DoVerb (OleContainer1.PrimaryVerb);
end;
```

Once a new object has been created, you can execute its primary verb using the DoVerb method. The program also displays a small toolbar with some bitmap buttons. I placed some TWinControl components in the form to let the user select them and thus disable the OleContainer. To keep this toolbar/panel visible while in-place editing is occurring, you should set its Locked property to True. This forces the panel to remain present in the application and not be replaced by a toolbar of the server.

To show what happens when you don't use this approach, I've added to the program a second panel, with some more buttons. Because I haven't set its Locked property, this new toolbar will be replaced with that of the active OLE server. When in-place editing launches a server application that displays a toolbar, that server's toolbar replaces the container's toolbar, as you can see in the lower part of Figure 20.11.

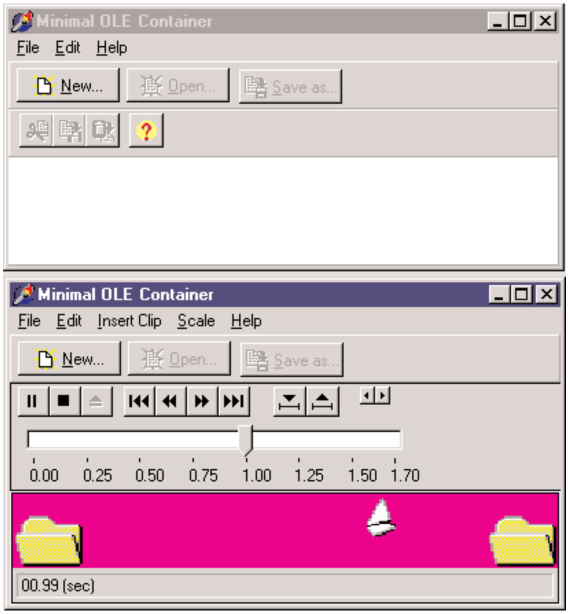
TIP

To make all the automatic resizing operations work smoothly, you should place the OLE container component in a panel component and align both of them to the client area of the form.

Another way to create an OLE object is to use the PasteSpecialDialog method, called in the PasteSpecial1Click event handler of the example. Another standard OLE dialog box, wrapped in a Delphi function, is the one showing the properties of the object, which is activated with the Object Properties item in the Edit pull-down menu by calling the ObjectPropertiesDialog method of the OleContainer component.

FIGURE 20.11:

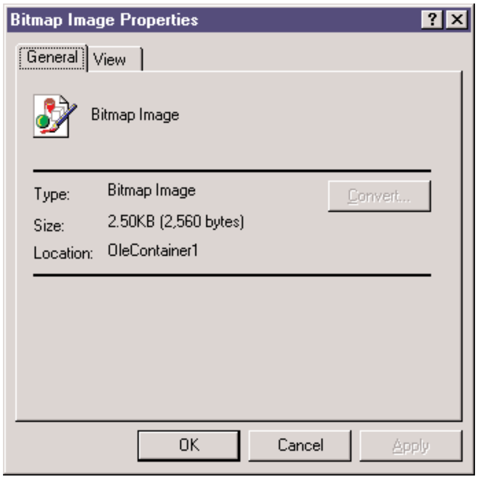
The second toolbar of the OleCont example (top) is replaced by the toolbar of the server (bottom).



You can see an example of the resulting standard OLE dialog box in Figure 20.12. Obviously, this dialog box changes depending on the nature of the active OLE object in the container. The last feature of the OleCont program is support for files; this is actually one of the simplest additions we can make, because the OLE container component already provides file support.

FIGURE 20.12:

The standard OLE Object Properties dialog box, available in the OleCont example



Type: Bitmap Image

Size: 2.50KB (2,560 bytes)

Location: OleContainer1

Using the Internal Object

In the preceding program, the user determined the type of the internal object created by the program. In this case, there is little you can do to interact with the internal objects. Suppose, instead, that you want to embed a Word document in a Delphi application and then modify it by code. You can do this by using OLE Automation with the embedded object, as demonstrated by the WordCont example (the name stands for *Word container*).

WARNING Since the WordCont example includes an object of a specific type, a Microsoft Word document, it won't run if you don't have that server application installed. Having a different version of the server might also create problems if the Automation methods used by the client program are not available in that version of the server.

In the form of this example, I've added an OleContainer component, set its `AutoActivate` property to aaManual (so that the only possible interaction is with our code), and added a toolbar with a couple of buttons. The code for the two buttons is quite straightforward, once you know that the embedded object corresponds to a Word document:

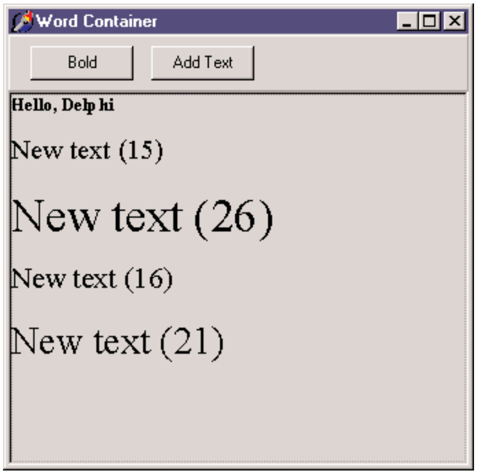
```
procedure TForm1.Button1Click(Sender: TObject);
var
  Document: Variant;
begin
  // activates if not running
  if not (OleContainer1.State = osRunning) then
    OleContainer1.Run;
  // get the document
  Document := OleContainer1.OleObject;
  // first paragraph to bold
  Document.Paragraphs.Item(1).Range.Bold := 1;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  Document, Paragraph: Variant;
begin
  // activate if not running
  if not (OleContainer1.State = osRunning) then
    OleContainer1.Run;
  // get the document
  Document := OleContainer1.OleObject;
  // add paragraphs, getting the last one
  Document.Paragraphs.Add;
  Paragraph := Document.Paragraphs.Add;
  // add text to the paragraph, using random font size
```

```
    Paragraph.Range.Font.Size := 10 + Random (20);
    Paragraph.Range.Text := 'New text (' +
      IntToStr (Paragraph.Range.Font.Size) + ')'#13;
  end;
```

You can see the effect of this code in Figure 20.13. The code is not terribly powerful, but it does show how you can merge the usage of OLE Containers and OLE Automation techniques.

FIGURE 20.13:

The WordCont example shows how to use OLE Automation with an embedded object.



Hello, Delp hi

New text (15)

New text (26)

New text (16)

New text (21)

Introducing ActiveX Controls

Microsoft's Visual Basic was the first program development environment to introduce the idea of supplying software components to the mass market. Actually, the concept of reusable software components is older than Visual Basic—it's well rooted in the theories of object-oriented programming (OOP). But OOP languages never delivered the reusability they promised, probably more because of marketing and standardization problems than for any other reason. Although Visual Basic does not fully exploit OOP, it applies the component concept through its standard way of building and distributing new controls that developers can integrate into the environment.

The first technical standard promoted by Visual Basic was *VBX*, a 16-bit specification that was fully available in the 16-bit version of Delphi. In moving to the 32-bit platforms, Microsoft replaced the VBX standard with the more powerful and more open *ActiveX* controls.

NOTE ActiveX controls used to be called OLE controls (or OCX). The name change reflects a new marketing strategy from Microsoft rather than a technical innovation. Technically, ActiveX can be considered a minor extension to the OCX technology. Not surprisingly, then, ActiveX controls are usually saved in files with the `.ocx` extension.

From a general perspective, an ActiveX control is not very different from a Windows, Delphi, or Visual Basic control. A control in any of these languages is always a window, with its associated code defining its behavior. The key difference between various families of controls is in the *interface* of the control—the interaction between the control and the rest of the application. Typical Windows controls use a message-based interface; VBX controls use properties and events; OLE Automation objects use properties and methods; and ActiveX controls use properties, methods, and events. These three elements of properties, methods, and events are also found in Delphi's own components.

Using OLE jargon, an ActiveX control is a "compound document object which is implemented as an in-process server DLL and supports OLE Automation, visual editing, and inside-out activation." Perfectly clear, right? Let's see what this definition actually means. An ActiveX control uses the same approach as OLE server objects, which are the objects you can insert into an OLE Document, as we saw in the last chapter. The difference between a generic OLE server and an ActiveX control is that, whereas ActiveX controls can only be implemented in one way, OLE servers can be implemented in three different ways:

- As stand-alone applications (for example, Microsoft Excel)
- As out-of-process servers—that is, executables files that cannot be run by themselves and can only be invoked by a server (for example, Microsoft Graph and similar applications)
- As in-process servers, such as DLLs loaded into the same memory space as the program using them

ActiveX controls can only be implemented using the last technique, which is also the fastest: as in-process servers. Furthermore, ActiveX controls are OLE Automation servers. This means you can access properties of these objects and call their methods. You can see an ActiveX control in the application that is using it and interact with it directly in the container application window. This is the meaning of the term *visual editing*, or *in-place activation*. A single click activates the control rather than the double-click used by OLE Documents, and the control is active whenever it is visible (which is what the term *inside-out activation* means), without having to double-click it.

As I've mentioned before, an ActiveX control has properties, methods, and events. Properties can identify states, but they can also activate methods. (This is particularly true for ActiveX controls that are *updated* VBX controls, because in a VBX there *was* no other way to activate a method than by setting a property.) Properties can refer to aggregate values, arrays, subobjects, and so on. Properties can also be dynamic (or read-only, to use the Delphi term).

In an ActiveX control, properties are divided into different groups: stock properties that most controls need to implement; ambient properties that offer information about the container (similar to the `ParentColor` or `ParentFont` properties in Delphi); extended properties managed by the container, such as the position of the object; and custom properties, which can be anything.

Events and methods are, well, events and methods. *Events* relate to a mouse click, a key press, the activation of a component, and other specific user actions. *Methods* are functions and procedures related to the control. There is no major difference between the ActiveX and Delphi concepts of events and methods.

ActiveX Controls Versus Delphi Components

Before I show you how to use and write ActiveX controls in Delphi, let's go over some of the technical differences between the two kinds of controls. ActiveX controls are DLL-based. This means that when you use them, you need to distribute their code (the OCX file) along with the application using them. In Delphi, the code of the components can be statically linked to the executable file or dynamically linked to it using a run-time package, so you can always choose.

Having a separate file allows you to share code among different applications, as DLLs usually do. If two applications use the same control (or run-time package), you need only one copy of it on the hard disk and a single copy in memory. The drawback, however, is that if the two programs have to use two different versions (or builds) of the ActiveX control, some compatibility problems might arise. An advantage of having a self-contained executable file is that you will also have fewer installation problems.

Now, what is the drawback of using Delphi components? The real problem is not that there are fewer Delphi components than ActiveX controls, but that if you buy a Delphi component, you'll only be able to use it in Delphi and Borland C++Builder. If you buy an ActiveX control, on the other hand, you'll be able to use it in multiple development environments from multiple vendors. Even so, if you develop mainly in Delphi and find two similar components based on the two technologies, I suggest you buy the Delphi one—it will be more integrated with your environment, and therefore easier for you to use. Also, the native Delphi component will probably be better documented (from the Pascal perspective), and it will take advantage of Delphi and Object Pascal features not available in the general ActiveX interface, which is traditionally based on C and C++.

Using ActiveX Controls in Delphi

Delphi comes with some preinstalled ActiveX controls, and you can buy and install more third-party ActiveX controls easily. After this description of how ActiveX controls work in general, I'll demonstrate one in an example.

The Delphi installation process is very simple. Select Component ➢ Import ActiveX Control in the Delphi menu. This opens the Import ActiveX dialog box, where you can see the list of ActiveX control libraries registered in Windows. If you choose one, Delphi will read its type library, list its controls, and suggest a filename for its unit. If the information is correct, click the Create Unit button to view the Pascal source code file created by Delphi as a *wrapper* for the ActiveX control. Click the Install button to add this new unit to a Delphi package and to the Component Palette.

Using the WebBrowser Control

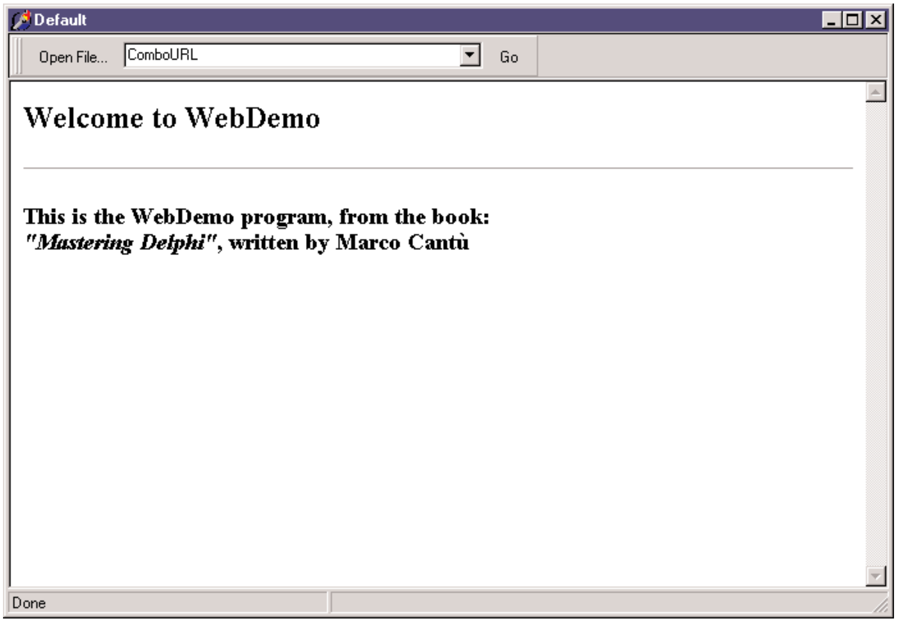
To build my example, I've used a preinstalled ActiveX control available in Delphi. Unlike the third-party controls, this is not available in the ActiveX page of the palette, but in the Internet page. The control, called WebBrowser, is a wrapper around Microsoft's Internet Explorer engine. The example is a very limited Web browser.

The WebBrows program on the CD-ROM has a `TWebBrowser` ActiveX control covering its client area and a control bar at the top and a status bar at the bottom. To move to a given Web page, a user can type in the combo box of the toolbar, select one of the visited URLs (saved in the combo box), or click on the Open File button to select a local file.

The actual implementation of the code used to select a Web or local HTML file is in the `GotoPage` method:

```
procedure TForm1.GotoPage(ReqUrl: string);
begin
  WebBrowser1.Navigate (ReqUrl, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam);
end;
```

`EmptyParam` is a predefined OleVariant you can use whenever you want to pass a default value as a reference parameter. This is a handy shortcut you can use to avoid creating an empty OleVariant each time you need a similar parameter. This method is called for by a file, when the user clicks on the Enter key in the combo box, or by selecting the Go button, as you can see in the source code on the companion CD.



Welcome to WebDemo

This is the WebDemo program, from the book:
"Mastering Delphi", written by Marco Cantù

FIGURE 20.14: The WebDemo program at startup: it fully supports graphics and all other Web extensions, as it is based on the Internet Explorer engine.

The program also handles four events of the WebBrowser control. When the download operations start and end, the program updates the text of the status bar and also the drop-down list of the combo box:

```
procedure TForm1.WebBrowser1DownloadBegin(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := 'Downloading ' +
    WebBrowser1.LocationURL + '...';
end;

procedure TForm1.WebBrowser1DownloadComplete(Sender: TObject);
var
  NewUrl: string;
begin
  StatusBar1.Panels[0].Text := 'Done';
  // add URL to combobox
  NewUrl := WebBrowser1.LocationURL;
  if (NewUrl <> '') and (ComboURL.Items.IndexOf (NewUrl) < 0) then
    ComboURL.Items.Add (NewUrl);
end;
```

Two other useful events are the `OnTitleChange`, used to update the caption with the title of the HTML document, and the `OnStatusTextChange` event, used to update the second part of the status bar. This code basically duplicates the information displayed in the first part of the status bar by the previous two event handlers.

Writing ActiveX Controls

Besides using existing ActiveX controls in Delphi, you can easily develop new ones. Although you can write the code of a new ActiveX control yourself, implementing all the required OLE interfaces (and there are many), it's much easier to use one of the techniques directly supported by Delphi:

- You can use the ActiveX Control Wizard to turn a VCL control into an ActiveX control. You start from an existing VCL component, which must be a `TWinControl` descendant, and Delphi wraps an ActiveX around it. During this step, Delphi adds a type library to the control. (Wrapping an ActiveX control around a Delphi component is exactly the opposite of what we did to use an ActiveX inside Delphi.)
- You can create an ActiveForm, place several controls inside it, and ship the entire form (without borders) as an ActiveX control. This second technique is the same one used by Visual Basic and is generally aimed at building Internet applications. However, it is also a very good alternative for the construction of an ActiveX control based on multiple Delphi controls or on Delphi components that do not descend from `TWinControl`.

An optional step you can take in both cases is to prepare a property page for the control, to use as a sort of property editor for setting the initial value of the properties of the control in any development environment—a kind of alternative to the Object Inspector in Delphi. Because most development environments allow only limited editing, it is more important to write a property page than it is to write a component or a property editor for a Delphi control.

Building an ActiveX Arrow

As an example of the development of an ActiveX control, I've decided to take the Arrow component we developed in Chapter 11, "Creating Components," and turn it into an ActiveX. We cannot use that component directly, because it was a graphical control, a subclass of `TGraphicControl`. However, turning a graphical control into a window-based control is usually a straightforward operation.

In this case, I've just changed the base class name to `TCustomControl` (and changed the name of the class of the control, as well, to avoid a name clash):

```
type
  TMdWArrow = class(TCustomControl)
  ...
```

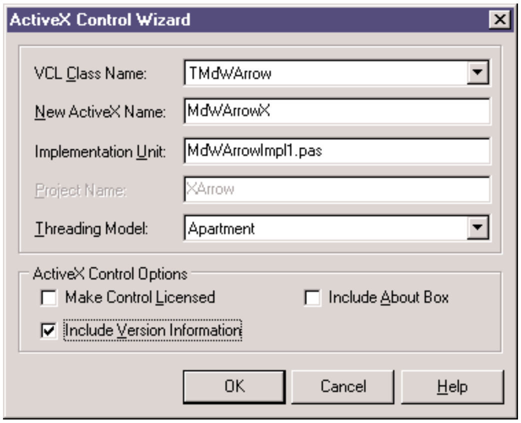
The `TWinControl` class has very minimal support for graphical output. Its `TCustomControl` subclass, however, has basically the same capabilities as the `TGraphicControl` class. The key difference is that a `TCustomControl` object has a window handle.

After installing this new component in Delphi, we are ready to start developing the new example. To create a new ActiveX library, select File ➢ New, move to the ActiveX page, and choose ActiveX library. Delphi creates the bare skeleton of a DLL, as we saw at the beginning of this chapter. I've saved this library as `XArrow`, in a directory with the same name, as usual.

Now it is time to use the ActiveX Control Wizard, available in the ActiveX page of the Object Repository—Delphi's New dialog box. In this wizard (shown in Figure 20.15), you select the VCL class you are interested in, customize the names shown in the edit boxes, and click OK; Delphi then builds the complete source code of an ActiveX control for you.

FIGURE 20.15:

Delphi's ActiveX Control Wizard



ActiveX Control Wizard

VCL Class Name: TMdWArrow

New ActiveX Name: MdWArrowX

Implementation Unit: MdWArrowImpl1.pas

Project Name: XArrow

Threading Model: Apartment

ActiveX Control Options

☐ Make Control Licensed

☐ Include About Box

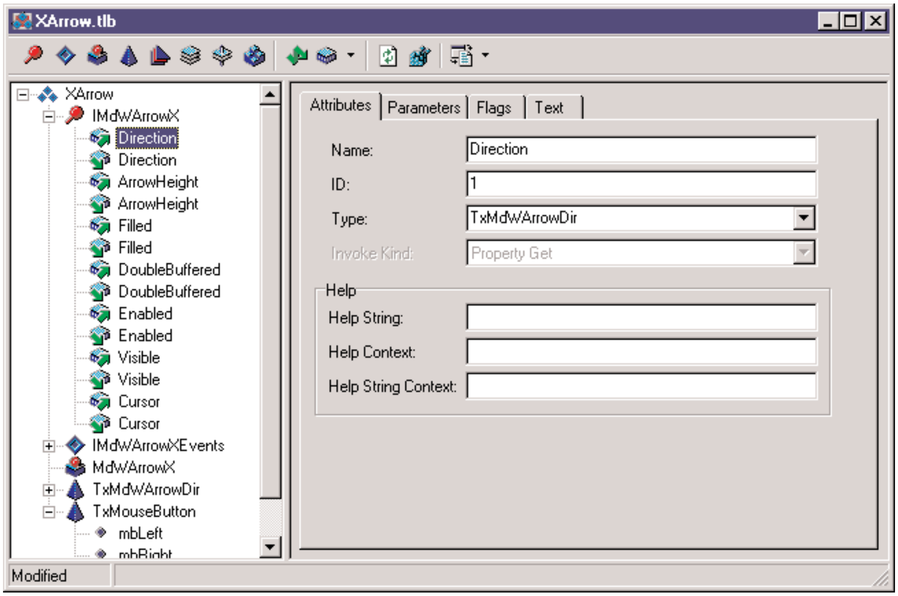
☑ Include Version Information

The use of the three check boxes at the bottom of the ActiveX Control Wizard window may not be obvious. If you include design-time license support, the user of the control won't be able to use it in a design environment without the proper *license key* for the control. The second check box allows you to include version information for the ActiveX, in the OCX file. If the third check box is selected, the ActiveX Control Wizard automatically adds an About box to the control.

Take a look at the code the ActiveX Control Wizard generates. The key element of this wizard is the generation of a type library. You can see the library generated for our arrow control in Delphi's type-library editor in Figure 20.16. From the type library information, the Wizard also generates an import file with the definition of an interface, the `dispinterface`, and other types and constants.

FIGURE 20.16:

The type-library editor with the type library of the demo ActiveX control I've created



Name: Direction
ID: 1
Type: TxMdWArrowDir
Invoke Kind: Property Get
Help String:
Help Context:
Help String Context:

In this example, the import file is named `XArrow_TLB.PAS`. The first part of this file includes a couple of GUIDs, one for the library as a whole and one for the control, and other constants for the definition of values corresponding to the OLE enumerated types used by properties of the Delphi control, for example:

```
type
  TxMdWArrowDir = TOleEnum;
const
  adUp = $00000000;
  adLeft = $00000001;
  adDown = $00000002;
  adRight = $00000003;
```

The real meat is the declaration of the `IMdWArrowX` interface, which I suggest you look at in the source code. Notice that the final part of the import unit includes the declaration of the `TMdWArrowX` class. This is a `TOleControl`-derived class you can use to install the control in Delphi, as we've seen in the first part of this chapter. You don't need this class to build the ActiveX control; you need it to install the ActiveX control in Delphi. The class used by the ActiveX server has the same class name but a different implementation.

The rest of the code, and the code you'll customize, is in the main unit, which in my example is called MdWArrowImpl1. This unit has the declaration of the ActiveX server object, `TMdWArrowX`, which inherits from `TActiveXControl` and implements the specific `IMdWArrowX` interface.

NOTE The TActiveXControl class does most of the work for providing ActiveX support in Delphi. This class implements interfaces required by every ActiveX control: IConnectionPointContainer, IDataObject, IObjectSafety, IOleControl, IOleInPlaceActiveObject, IOleInPlaceObject, IOleObject, IPerPropertyBrowsing, IPersistPropertyBag, IPersistStorage, IPersistStreamInit, IQuickActivate, ISimpleFrameSite, ISpecifyPropertyPages, IViewObject, and IViewObject2. Just the declaration of the TActiveXControl class takes more than 250 lines of code, and its implementation code is responsible for a good part of the 4,000 lines of code of the AxCtrls unit.

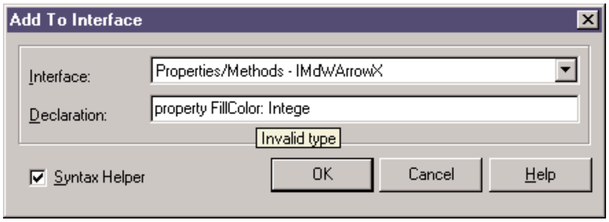
Before we customize this control in any way, let's see how it works. You should first compile the ActiveX library and then register it using Delphi's Run ➢ Register ActiveX Server menu command. Now you can install the ActiveX control as we've done in the past, except you have to specify a different name for the new class to avoid a name clash. If you use this control, it doesn't look much different from the original VCL control, but the advantage is that the same component can now be installed also in other development environments.

Adding New Properties

Once you've created an ActiveX control, adding new properties, events, or methods to it is—surprisingly—simpler than doing the same operation for a VCL component. Delphi, in fact, provides specific visual support for the former, not for the latter.

You can open the Pascal unit with the implementation of the ActiveX control, and choose Edit ➢ Add To Interface. As an alternative, you can use the same command from the shortcut menu of the editor. Delphi opens the Add To Interface dialog box (see Figure 20.17). In the combo box of this dialog box, you can choose between a new property, method, or event. In this example, the first selection will affect the IMdWArrowX interface and the second the IMdWArrowXEvents interface.

FIGURE 20.17: The Add To Interface dialog box, with the syntax helper in action


Add To Interface

Interface: Properties/Methods - IMdWArrowX

Declaration: property FillColor: Intege

Invalid type

☑ Syntax Helper

In the edit box, you can then type the declaration of this new interface element. If the Syntax Helper check box is activated, you'll get hints describing what you should type next and highlighting any errors. You can see the syntax helper in action in Figure 20.17. When you define a new ActiveX interface element, keep in mind that you are restricted to OLE data types. In the XArrow example, I've added two properties to the ActiveX control. Because the Pen and the Brush properties of the original Delphi components are not accessible, I've made their color available. These are examples of what you can write in the edit box of the Add To Interface dialog (executing it twice):

```
property FillColor: Integer;
property PenColor: Integer;
```

NOTE Since a TColor is a specific Delphi definition, it is not legal to use it. TColor is an Integer subrange that defaults to Integer size, so I've used the standard Integer type directly.

The declarations you enter in the Add To Interface dialog box are automatically added to the control's type library (TLB) file, to its import library unit, and to its implementation unit:

```
type
  IMdWArrowX = interface(IDispatch)
    function Get_FillColor: Integer; safecall;
    procedure Set_FillColor(Value: Integer); safecall;
    function Get_PenColor: Integer; safecall;
    procedure Set_PenColor(Value: Integer); safecall;
    ...
    property FillColor: Integer read Get_FillColor write Set_FillColor;
    property PenColor: Integer read Get_PenColor write Set_PenColor;
```

All you have to do to finish the ActiveX control is fill in the Get and Set methods of the implementation. Here is the code of the first property:

```
function TMdWArrowX.Get_FillColor: Integer;
begin
  Result := ColorToRGB (FDelphiControl.Brush.Color);
end;

procedure TMdWArrowX.Set_FillColor(Value: Integer);
begin
  FDelphiControl.Brush.Color := Value;
end;
```

If you now install this ActiveX control in Delphi once more, the two new properties will appear. The only problem with this property is that Delphi uses a plain integer editor, making it quite difficult to enter the value of a new color by hand. A program, by contrast, can easily use the RGB function to create the proper color value.

Adding a Property Page

As it stands, other development environments can do very little with our component, because we've prepared no property page—no property editor. A property page is fundamental so that programmers using the control can edit its attributes. However, adding a property page is not as simple as adding a form with a few controls. The property page, in fact, will integrate with the host development environment. The property page for our control will show up inside a property page dialog of the host environment, which will provide the OK, Cancel, and Apply buttons, and the tabs for showing multiple property pages (some of which might be provided by the host environment).

The nice thing is that support for property pages is built into Delphi, so adding one takes little time. You open an ActiveX project, then open the usual New Items dialog box, move to the ActiveX page, and choose Property Page. What you get is not very different from a form. In fact, the `TPropertyPage1` class (created by default) inherits from the `TPropertyPage` class of VCL, which in turn inherits from `TCustomForm`.

TIP Delphi provides four built-in property pages for colors, fonts, pictures, and strings. The GUIDs of these classes are indicated by the constants `Class_DColorPropPage`, `Class_DFontPropPage`, `Class_DPicturePropPage`, and `Class_DStringPropPage` in the AxCtrls unit.

In the property page, you can add controls as in a normal Delphi form, and you can write code to let the controls interact. I've added to the property page a combo box with the possible values of the `Direction` property, a check box for the `Filled` property, an edit box with an UpDown control to set the `ArrowHeight` property, and two shapes with corresponding buttons for the colors. The only code added to the form relates to the two buttons used to change the color of the two shape components, which offer a preview of the colors of the actual ActiveX control. The `OnClick` event of the button uses a ColorDialog component, as usual:

```
procedure TPropertyPage1.ButtonPenClick(Sender: TObject);
begin
  with ColorDialog1 do
  begin
    Color := ShapePen.Brush.Color;
    if Execute then
    begin
      ShapePen.Brush.Color := Color;
      Modified; // enable Apply button!
    end;
  end;
end;
```

What is important to notice in this code is the call to the `Modified` method of the `TPropertyPage` class. This call is required to let the property page dialog box know we've modified one of the values and to enable the Apply button. When a user interacts with one of the other controls of this form, this call is made automatically. For the two buttons, however, we need to add this line ourselves.

TIP Another tip relates to the Caption of the property page form. This will be used in the property dialog box of the host environment as the caption of the tab corresponding to the property page.

The next step is to associate the controls of the property page with the actual properties of the ActiveX control. The property page class automatically has two methods for this: UpdateOleObject and UpdatePropertyPage. As their names suggest, these two methods copy data from the property page to the ActiveX control and vice versa. Here is the code for my example:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
  { Update your controls from the OleObject }
  ComboDir.ItemIndex := OleObject.Direction;
  CheckFilled.Checked := OleObject.Filled;
  EditHeight.Text := IntToStr (OleObject.ArrowHeight);
  ShapePen.Brush.Color := OleObject.PenColor;
  ShapePoint.Brush.Color := OleObject.FillColor;
end;

procedure TPropertyPage1.UpdateObject;
begin
  { Update the OleObject from your controls }
  OleObject.Direction := ComboDir.ItemIndex;
  OleObject.Filled := CheckFilled.Checked;
  OleObject.ArrowHeight := UpDownHeight.Position;
  OleObject.PenColor := ColorToRGB (ShapePen.Brush.Color);
  OleObject.FillColor := ColorToRGB (ShapePoint.Brush.Color);
end;
```

The final step is to connect the property page itself to the ActiveX control. When the control was created, the Delphi ActiveX Control Wizard automatically added a declaration for the DefinePropertyPages method to the implementation unit. In this method, we call the DefinePropertyPage method (this time the method name is singular) for each property page we want to add to the ActiveX. This method has as its parameter the GUID of the property page, something you can find in the corresponding unit. (Of course, you'll need to add a uses statement referring to that unit.) Here is the code of my example:

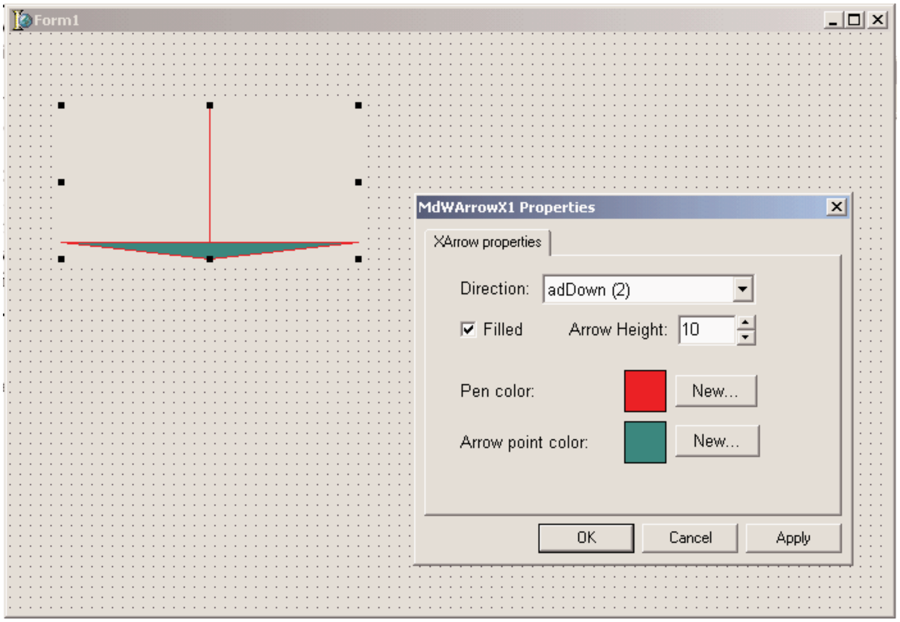
```
procedure TMdWArrowX.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

NOTE The connection between the ActiveX control and its property page takes place using a GUID. This is possible because the property page object can be created through a class factory, and its GUID is stored in the Windows Registry when you register the ActiveX control library. To see what's going on, look at the `initialization` section of the property page unit, which calls `TActiveXPropertyPageFactory.Create`.

Now that we've finished developing the property page, and after recompiling and reregistering the ActiveX library, we can install the ActiveX control inside a host development environment (including Delphi itself) and see how it looks. Figure 20.18 shows an example. (If you've already installed the ActiveX control in Delphi, you should uninstall it prior to rebuilding it. This process might also require closing and reopening Delphi itself.)

FIGURE 20.18:

The XArrow ActiveX control and its property page, hosted by the Delphi environment



Direction: adDown (2)

☑ Filled

Arrow Height: 10

Pen color: 🟥 New...

Arrow point color: 🟩 New...

ActiveForms

As I've mentioned, Delphi provides an alternative to the use of the ActiveX Control Wizard to generate an ActiveX control. You can use an ActiveForm, which is an ActiveX control that is based on a form and can host one or more Delphi components. This is exactly the technique used in Visual Basic to build new controls, and it makes sense when you want to create a compound component.

For example, to create an ActiveX clock, we can place on an ActiveForm a label (a graphic control that cannot be used as a starting point for an ActiveX control) and a timer, and connect the two with a little code. The form/control becomes basically a container of other controls, which makes it very easy to build compound components (easier than for a VCL compound component).

To build such a control, close the current project, and select the ActiveForm icon in the ActiveX page of the File ➢ New dialog box. Delphi asks you for some information in the following ActiveForm Wizard dialog box, similar to the ActiveX Control Wizard dialog box.

ActiveForm Internals

Before we continue with the example, let's look at the code generated by the ActiveForm Wizard. The key difference from a plain Delphi form is in the declaration of the new form class, which inherits from the TActiveForm class and implements a specific ActiveForm interface:

```
type
  TAXForm1 = class(TActiveForm, IAXForm1)
```

As usual, the IAXForm interface is declared in the type library and in a corresponding Pascal file generated by Delphi. Here is a small excerpt of the IAXForm1 interface from the XF1Lib.pas file, with some comments I've added:

```
type
  IAXForm1 = interface(IDispatch)
    ['{51661AA1-9468-11D0-98D0-444553540000}']
    // Get and Set methods for TForm properties
    function Get_Caption: WideString; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    ...
    // TForm methods redeclared
    procedure Close; safecall;
    ...
    // TForm properties
    property Caption: WideString read Get_Caption write Set_Caption;
```

The code generated for the TAXForm1 class implements all the Set and Get methods, which change or return the corresponding properties of the form, and it implements the events, which again are the events of the form. Here is a small excerpt:

```
  private
    procedure ActivateEvent(Sender: TObject);
  protected
    procedure Initialize; override;
    function Get_Caption: WideString; safecall;
    procedure Close; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
```

Let's look at the implementation of properties first:

```
function TAXForm1.Get_Caption: WideString;
```

```
begin
  Result := WideString(Caption);
end;

procedure TAXForm1.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;
```

The TForm events are set to the internal methods when the form is created:

```
procedure TAXForm1.Initialize;
begin
  OnActivate := ActivateEvent;
  ...
end;
```

Each event then maps itself to the external ActiveX event, as in the following two methods:

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;
```

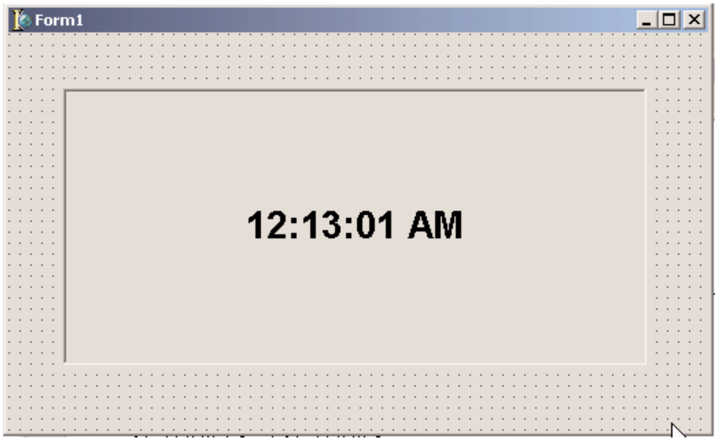
Because of this mapping, you should not handle the events of the form directly. Instead, you can either add some code to these default handlers or override the TForm methods that end up calling the events. (This is exactly the approach you use when building a Delphi component.) Keep in mind that the interface properties of an ActiveForm are meant for developers using it as a control, not for final users of the ActiveForm on the Web. This mapping problem refers only to the events of the form itself, not to the events of the components of the form. You can continue to handle the events of the components as usual.

The XClock ActiveX Control

Now that we've looked at the code generated by Delphi, we can return to the development of the XClock example. Place on the form a label with a large font and centered text, aligned to the client area and a timer. Then write an event handler for its OnTimer event, so that the control updates the output of the label with the current time every second:

```
procedure TXClock.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr (Time);
end;
```

Now compile this library, register it, and install it in a package to test it in the Delphi environment. You can see an example of its use in Figure 20.19. Notice in this figure the effect of the sunken border. This is controlled by the AxBorderStyle property of the active form, one of the few properties of active forms that is not available for a plain form.



12:13:01 AM

FIGURE 20.19:
The ActiveX timer installed in a Delphi package

ActiveForms are usually considered as a technique to deploy a Delphi application via the Internet. However, the ActiveX and ActiveForm support provided by Delphi represent to different ways to build ActiveX controls, which can be used both on a Web page and in another development environment.

ActiveX in Web Pages

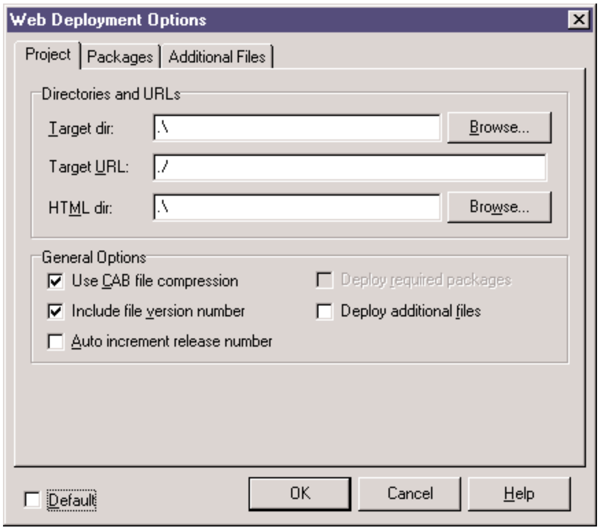
In the last example, we used Delphi's ActiveForm technology to create a new ActiveX control. In fact, an ActiveForm is an ActiveX control based on a form. Borland documentation often implies that ActiveForms should be used in HTML pages, but you can use any ActiveX control on a Web page.

NOTE Microsoft once promoted ActiveX as an Internet technology for delivering interactive content. Due to complexities and security problems inherent in downloading executable code, the market never really bought into this. Microsoft has since dropped ActiveX from its Internet technologies list. Still, this technology might have a value in an intranet to let you deliver small applications to users of your local area network, as you can relax the security settings when accessing local Web sites.

Basically, each time you create an ActiveX library, Delphi enables the Project ➢ Web Deployment Options and Project ➢ Web Deploy menu items. The first allows you to specify how and where to deliver the proper files. As shown in Figure 20.20, in this dialog box you can set the server directory for deploying the ActiveX component, the URL of this directory, and the server directory for deploying the HTML file (which will have a reference to the ActiveX library using the URL you provide).

FIGURE 20.20:

The Web Deployment Options dialog box



Directories and URLs

Target dir: .\

Target URL: ./

HTML dir: .\

General Options

☑ Use CAB file compression

☑ Include file version number

□ Auto increment release number

□ Deploy required packages

□ Deploy additional files

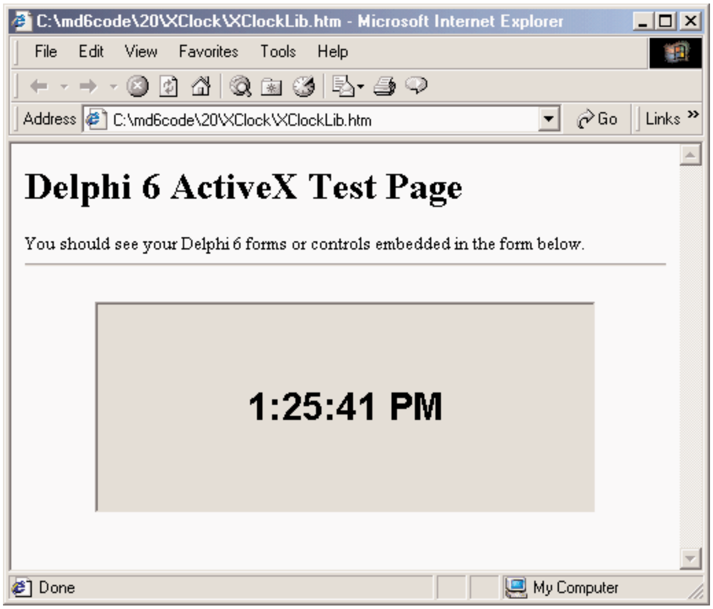
□ Default

You can also specify the use of a compressed CAB file, which can store the OCX file and other auxiliary files, such as packages, making it easier and faster to deliver the application to the user. A compressed file, in fact, means a faster download. Using the options shown in Figure 20.20, Delphi generates the HTML file and CAB file for the XClock project in the same directory. Opening this HTML file in Internet Explorer produces the output shown in Figure 20.21.

WARNING At times, when you load an HTML page referring to an ActiveX, all you get is a red *X* marker indicating a failure to download the control. There are various possible explanations for this problem. First, Internet Explorer must be set up properly, allowing the download of controls and (if the control is not signed) lowering the security level. Second, other problems might arise when the control requires a DLL or a package that is not part of the downloaded CAB file. Third, you might get the red slash marker when there is a mismatch in the version number—*or* you might see an older version of the control in action.

FIGURE 20.21:

The XClock control in the sample HTML page


C:\md6code\20\XClock\XClockLib.htm - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address C:\md6code\20\XClock\XClockLib.htm
Go
Links
Done
My Computer

Delphi 6 ActiveX Test Page

You should see your Delphi 6 forms or controls embedded in the form below.

1:25:41 PM

Besides showing you how to deploy the XClock control on a Web page, I've created the XForm1 example to demonstrate the problems with event handlers of ActiveForms mentioned in the previous section "ActiveForm Internals." Because the form events are exported as events of the control, you should not handle the events of the form directly but add some code to the default handlers provided by the Active Form. For example, if you add a handler for the `OnPaint` event of the form and write the following code, it will never be executed:

```
procedure TFormX1.FormPaint(Sender: TObject);
begin
  Canvas.Brush.Color := clYellow;
  Canvas.Ellipse(0, 0, Width, Height);
end;
```

If you want to paint something on the form's background, instead, you have to modify the corresponding handler installed by the ActiveForm Wizard:

```
procedure TFormX1.PaintEvent(Sender: TObject);
begin
  Canvas.Brush.Color := clBlue;
  Canvas.Rectangle (20, 20, ClientWidth - 20, ClientHeight - 20);
  if FEvents <> nil then FEvents.OnPaint;
end;
```

As an alternative, you can place a frame, a panel, or another component on the surface of the form, and handle *its* events. In the XForm1 example, I've added a PaintBox component, with a bevel component behind it to make the area of the PaintBox visible.

The Role of an ActiveX Form on a Web Page

Before we look at another example, it is important to stop for a second to consider the role of an ActiveX form placed inside a Web page. Basically, placing a form in a Web page corresponds to letting a user download and execute a custom Windows application. There is little else happening. You download an executable file and start it. This is one of the reasons the ActiveX technology raises so many concerns about security.

The XFUser example highlights the situation. It calls the `GetUserName` Windows API function and shows the user name on the screen. Its effect is certainly not astonishing, as the name of the user will be displayed in a label. However, this example highlights a couple of important points (which apply both to ActiveForms and ActiveX controls in general):

- In an ActiveX control or form, you can call any Windows API function (which means the user viewing the Web page must have Windows on his or her computer) or certain Windows API–compatible libraries.
- An ActiveX can access the system information of the computer, such as the user name, the directory structure, and so on. This is why, before downloading an ActiveX, Web browsers check whether the ActiveX has a proper authentication, or signature. (You should note that this signature identifies the author of the control and that the module has not been corrupted or tampered with since the author published it; it doesn't prove in any way that the control is safe.)

Well, I could continue, but I think my point is clear. ActiveX controls and ActiveForms inside Web pages have problems, and even Microsoft has slowly abandoned this technology. For this reason, I'm going to show you only one more example, which is instructive on how external environments can interact with an ActiveX control.

Setting Properties for the XArrow

An ActiveForm has a few properties you can set when you use it inside a development environment, and a plain ActiveX control has even more. For example, if you want to set properties in the HTML file hosting the control, you can use a special `param` tag, but the control must support a special interface known as `IPersistPropertyBag`.

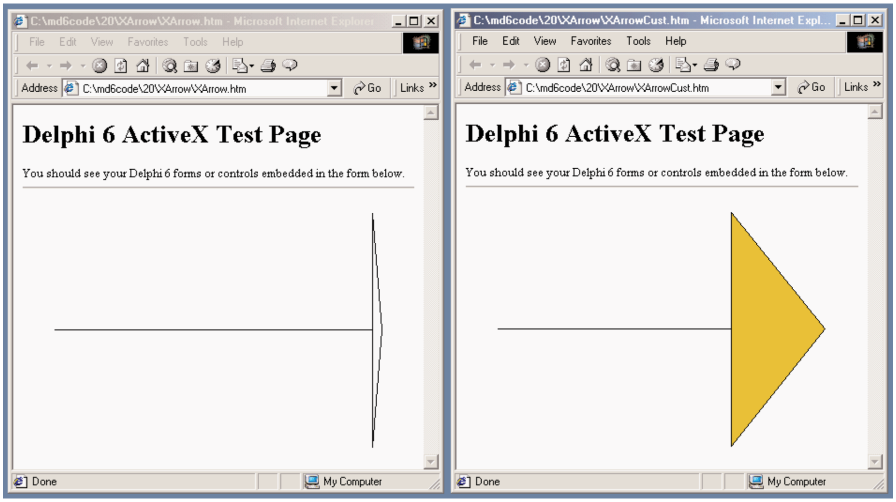
Starting with Delphi 4, the `IPersistPropertyBag` support is built in, providing support for all of the properties of the ActiveX control or ActiveForm. As an example, I've used the Web Deploy options on the XArrow control. Then, I've modified the automatically generated HTML file with three `param` tags:

```
<object classid="clsid:482B2145-4133-11D3-B9F1-00000100A27B"
  codebase="./XArrow.cab" width="350" height="250" align="center"
  hspace="0" vspace="0">
  <param name="ArrowHeight" value="100">
  <param name="Filled" value="-1">
  <param name="FillColor" value="111829">
</object>
```

You can compare the default and customized output of the control in Figure 20.22.

FIGURE 20.22:

By using the `param` tag, we can set values for the properties of an ActiveX control in the HTML file hosting it. The two copies of the program show the default and the customized output.



C:\md6code\20\XArrow\XArrow.htm - Microsoft Internet Explorer

Address C:\md6code\20\XArrow\XArrow.htm

Delphi 6 ActiveX Test Page

You should see your Delphi 6 forms or controls embedded in the form below.

C:\md6code\20\XArrow\XArrowCust.htm - Microsoft Internet Expl...

Address C:\md6code\20\XArrow\XArrowCust.htm

Delphi 6 ActiveX Test Page

You should see your Delphi 6 forms or controls embedded in the form below.

Introducing COM+

In addition to plain COM servers, Delphi also allows you to create enhanced COM objects, including stateless objects and transaction support. This type of COM object was first introduced by Microsoft with the MTS (Microsoft Transaction Server) acronym, and later renamed as COM+ in Windows 2000.

Delphi 6 supports building both standard stateless objects and DataSnap remote data modules based on stateless objects. In both cases you'll start the development by using one of the available Delphi wizards, using the New Items dialog box and selecting the Transactional Object icon of the ActiveX page or the Transactional Data Module icon of the Multitier page. You must add these objects to an ActiveX library project, not to a plain application. Another icon, COM+ Event Object, is used to support COM+ events.

MTS is an operating-system service you can install on Windows NT and 98; it was renamed as COM+ in Windows 2000, so I'll call it COM+ but this actually refers to both. This system service provides a run-time environment supporting database transaction services, security, resource pooling, and an overall improvement in robustness for DCOM applications. The run-time environment manages objects called *COM+ components*. These are COM objects stored in an in-process server (that is, a DLL). While other COM objects run directly in the client application, COM+ objects are handled by this run-time environment, in which you install the COM+ libraries. COM+ objects must support specific COM interfaces, starting with `IObjectControl`, which is the base interface (like `IUnknown` for a COM object).

Before getting into too many technical and low-level details, let's consider COM+ from a different perspective. What are the benefits of this approach? COM+ provides a few interesting features, including:

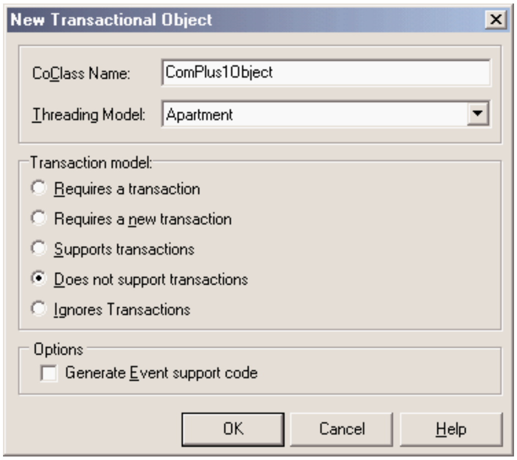
Role-based security The role assigned to a client determines whether it has the right to access the interface of a data module.

Reduced database resources You can reduce the number of database connections, as the middle tier logs on to the server and uses the same connections for multiple clients (although you cannot have more clients connected at once than you have licenses for the server).

Database transactions COM+ transaction support includes operations on multiple databases, although few SQL servers other than Microsoft's support COM+ transactions.

Creating a COM+ Component

The starting point for creating a COM+ component is the creation of an ActiveX library project. After this step you can select a new Transactional Object in the ActiveX page of the New Items dialog box. In the resulting dialog box (see Figure 20.23), enter the name of the new component (*ComPlus1Object* in my ComPlus1 example).



CoClass Name: ComPlus1Object

Threading Model: Apartment

Transaction model:

- ○ Requires a transaction
- ○ Requires a new transaction
- ○ Supports transactions
- ● Does not support transactions
- ○ Ignores Transactions

Options

- □ Generate Event support code

FIGURE 20.23: Delphi's New Transactional Object dialog box, used to create a COM+ object

The New Transactional Object dialog box allows you to enter a name for the class of the COM+ object, the threading model (because COM+ serializes all the requests, Single or Apartment will generally do), and a transactional model:

> **Requires A Transaction** indicates that each call from the client to the server is considered to be a transaction (unless the caller supplies an existing transaction context).
>
> **Requires A New Transaction** indicates that each call is considered a new transaction.
>
> **Supports Transactions** indicates that the client must explicitly provide a transaction context.
>
> **Does Not Support Transaction** (the default choice, and the one I've used) indicates that the remote data module won't be involved in any transaction.

As you close this dialog, Delphi adds a type library and an implementation unit to the project and opens the type-library editor, where you can define the interface of your new COM object. For this example I've added a `Value` integer property, an `Increase` method having as parameter an amount, and an `AsText` method returning a WideString with the formatted value. As you accept the edits in the type-library editor (clicking the Refresh button or closing the window), Delphi shows the Implementation File Update Wizard, if the corresponding IDE option is set. This wizard will ask for your confirmation before adding four methods to the class, including the get and set methods of the property. You can now write some code for the COM object, which for my example was quite trivial.

"Ignores Transactions" indicates that objects do not participate in transactions, regardless of whether the □ client has a transaction. The difference from the setting "Does not support transactions" prevents the object □ from being activated if the client has a transaction.

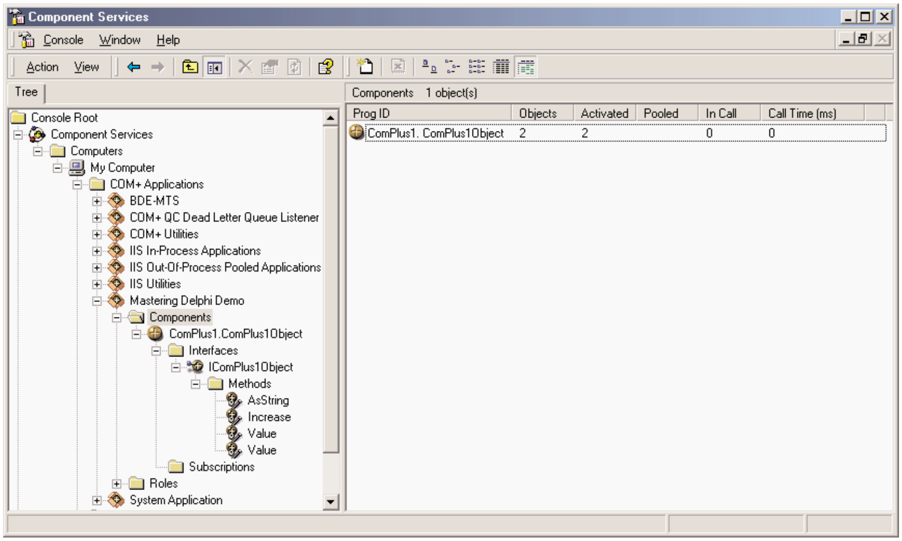
As you've compiled an ActiveX library, or COM library, which hosts a COM+ component, you can use the Component Services administrative tool (shown in the Microsoft Management Console, or MMC) to install and configure the COM+ component. Even better, you can use the Delphi IDE to install the COM+ component using the Run ➢ Install COM+ Object menu command. In the subsequent dialog box, you'll be able to select the component to install (as a library can host multiple components), and choose the COM+ application where to install the component.

A COM+ application is nothing more than a way to group COM+ components; it is not an actual program or anything like one (why they call it application is not fully clear to me). So in the Install COM+ Object dialog, you can select an existing application/group, choose the Install Into New Application page, and enter a name and description for it.

I've called the COM+ application *Mastering Delphi Demo*, as you can see in Figure 20.24 in the Component Services administration. This is the front end you can use to fine-tune the behavior of your COM+ components, setting their activation model (just-in-time activation, object pooling, and so on), their transaction support, and the security and concurrency models you want to use. You can also use this console to monitor the objects and the actual method calls (in case these take a long time to execute). In Figure 20.24, you can see that there are currently two active objects.

FIGURE 20.24:

The newly installed COM+ component inside a custom COM+ application (as shown by Microsoft's Component Services tool)



Prog ID	Objects	Activated	Pooled	In Call	Call Time (ms)
ComPlus1. ComPlus1Object	2	2		0	0

TIP Because you've created one or more objects, the COM library remains loaded in the COM+ environment and some of the objects might be kept in cache, even if there are no clients connected to them. For this reason, you cannot generally recompile the COM library after using it, unless you use the MMC to shut it down.

I've actually created a client program for the COM+ object, but this is exactly like any other Delphi COM client. After importing the type library, which is automatically registered while installing the component, I created an interface-type variable referring to it and called its methods as usual. You can find the example on the CD accompanying this book.

Transactional Data Modules

The same types of features are available when creating a *transactional data module*—that is, a remote data module within a COM+ component. Once you've created a transactional data module, you can build a Delphi DataSnap application as we've done in Chapter 17, "Multitier Database Applications with DataSnap." You can add one or more dataset components, add one or more providers, and export the provider(s). You can also add custom methods to the data module type library by editing the type library or using the Add To Interface command.

Within a COM+ component or transactional data module, you can also use the `GetObjectContext` method, which returns the `IObjectContext` interface of the COM+ object. The `IObjectContext` interface provides support for transactions:

- You can use `SetComplete` to tell the COM+ environment that the object has finished working and can be deactivated, so that the transaction can be committed.
- You can call `EnableCommit` to indicate that the object hasn't finished but the transaction should be committed.
- You can call `DisableCommit` to stop the commit operation, even if the method is done, disabling the object deactivation between method calls.
- You can call `SetAbort` to say that the object has finished and can be activated but the transaction cannot be committed.
- You can call `IsInTransaction` to check whether the object is part of a transaction.

Other methods of the `IContextObject` interface include `CreateInstance`, which creates another COM+ object in the same context and within the current transaction, `IsCallerInRole`, which checks if the object's caller is in a particular "security" role, and `IsSecurityEnabled` (whose name is self-explanatory).

Once you've built a transactional data module within a server library, you can install it as I've shown above for a plain COM+ object. After the transactional data module has been installed, it will be directly available to other applications and visible in the management console.

An important feature of COM+ is that it becomes much easier to configure DCOM support using this environment. In fact, the COM+ environment of a client computer can grab information from the COM+ environment of a server computer, including registration information for the COM+ object you want to be able to call over a network. The same network configuration is way more complex if done with plain DCOM, without MTS or COM+.

TIP Even though COM+ configuration is much better than DCOM configuration, still you are limited to computers with a recent version of the Windows operating system. Considering that even Microsoft is moving away from DCOM technology, before you build a large system based on this technology you should at least evaluate the alternative provided by SOAP (discussed in Chapter 23, "XML and SOAP").

COM+ Events

Client applications that use traditional COM objects and Automation servers can call methods of those servers, but this is not an efficient way to check whether the server has updated data for the client. For this reason, it is possible for a client to define a COM object that implements a *callback* interface, pass this object to the server, and let the server call it. Traditional COM events (which use the `IConnectionPoint` interface) are simplified by Delphi for Automation objects, but are still quite complex to handle.

COM+ introduces a simplified event model, in which the events are COM+ components and the COM+ environment manages the connections. In traditional COM callbacks, the server object doesn't have to keep track of the multiple clients it has to notify to, which is one of the reasons for the complexity of its code. In COM+, the server calls into a single event interface, and the COM+ environment will forward the event to all clients that have expressed interest for it. This way, the client and the server are less coupled, making it possible for a client to receive notification from different servers, without any change in its code.

NOTE Some critics say that Microsoft introduced this model only because it was very complex to handle COM events in the traditional way for Visual Basic developers. Windows 2000 actually provides a few operating-system features specifically intended for VB developers.

To create a COM+ event, you should create a COM library (or ActiveX library) and use the COM+ Event Object wizard. The resulting project will contain a type library with the definition of the interface used to fire the events, plus some *fake* implementation code. The actual server that will receive the notification of the events, in fact, will provide the actual implementation of the interface. The fake code is there only to support Delphi's COM registration system.

While building the MdComEvents library, I added to the type library a single method with two parameters, resulting in the following code (in the interface definition file):

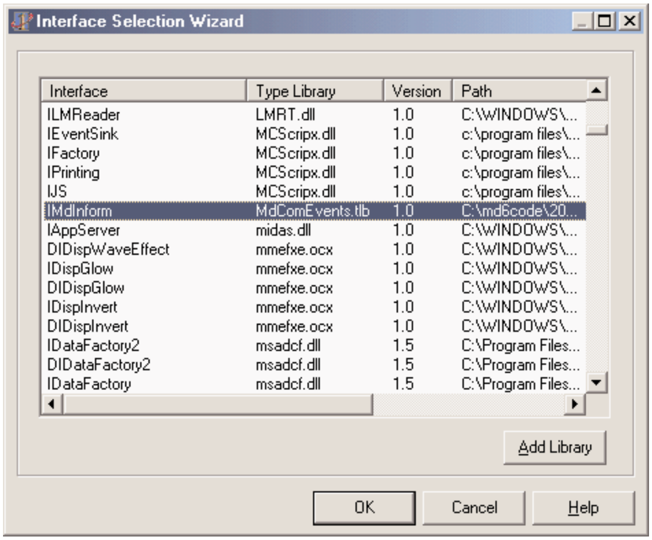
```
type
  IMdInform = interface(IDispatch)
    ['{202D2CC8-8E6C-4E96-9C14-1FAAE3920ECC}']
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;
```

The main unit includes the fake COM object and its class factory, to let the server register itself. The code looks like this (notice that the method is `abstract`, and it has no implementation):

```
type
  // fake abstract class
  TMdInform = class (TAutoObject, IMdInform)
  protected
    procedure Informs(Code: Integer; const Message: WideString);
      virtual; safecall; abstract;
  end;

begin
  TAutoObjectFactory.Create(ComServer, TMdInform, Class_MdInform,
    ciMultiInstance, tmApartment);
end.
```

At this point, you can compile the library and install it in the COM+ environment. Again, after selecting a COM+ application (that is, a group of COM+ components), use the shortcut menu of its Components folder to add a new component to it. In the COM Component Install Wizard, click the Install New Event Class button and select the library you've just compiled. Your COM+ event definition will be automatically installed.

To test whether it works, you'll have to build an actual implementation of this event interface and a client invoking it. The implementation can be added to another ActiveX library, hosting a plain COM object. Within Delphi's COM Object Wizard you can select the interface to implement, choosing it in the list that appears when you select the List button. An example of this rather long list, dubbed Interface Selection Wizard, is shown in Figure 20.25.



Interface	Type Library	Version	Path
ILMReader	LMRT.dll	1.0	C:\WINDOWS\...
IEventSink	MCScripx.dll	1.0	c:\program files\...
IFactory	MCScripx.dll	1.0	c:\program files\...
IPrinting	MCScripx.dll	1.0	c:\program files\...
IJS	MCScripx.dll	1.0	c:\program files\...
IMdInform	MdComEvents.tlb	1.0	C:\md6code\20...
IAppServer	midas.dll	1.0	C:\WINDOWS\...
DIDispWaveEffect	mmefxe.ocx	1.0	C:\WINDOWS\...
IDispGlow	mmefxe.ocx	1.0	C:\WINDOWS\...
DIDispGlow	mmefxe.ocx	1.0	C:\WINDOWS\...
IDispInvert	mmefxe.ocx	1.0	C:\WINDOWS\...
DIDispInvert	mmefxe.ocx	1.0	C:\WINDOWS\...
IDataFactory2	msadcf.dll	1.5	C:\Program Files...
DIDataFactory2	msadcf.dll	1.5	C:\Program Files...
IDataFactory	msadcf.dll	1.5	C:\Program Files...

FIGURE 20.25:

Delphi 6's new Interface Selection Wizard, used in this case to select an event interface

The resulting library, which in my example is called EvtSubscriber, exposes an Automation object, a COM object implementing the `IDispatch` interface (which is mandatory for COM+ Events). In my example, the object has the following definition and code:

```
type
  TInformSubscriber = class(TAutoObject, IMdInform)
  protected
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;

procedure TInformSubscriber.Informs(Code: Integer; const Message: WideString);
begin
  ShowMessage ('Message <' + IntToStr (Code) + '>: ' + Message);
end;
```

After compiling this library, you can first install it into the COM+ environment, then you have to bind it to the event. This second step is accomplished in the Component Services management console by selecting the `Subscriptions` folder under the event object registration, and using the New ➢ Subscription shortcut menu. In the resulting wizard, you can choose the interface to implement (but there is probably only one interface in your COM+ event library), then you'll see a list of COM+ components that implement this interface. Selecting one or more of them you'll set up the subscription binding, which is listed under the `Subscriptions` folder. You can see an example of my configuration while building this example in Figure 20.26.

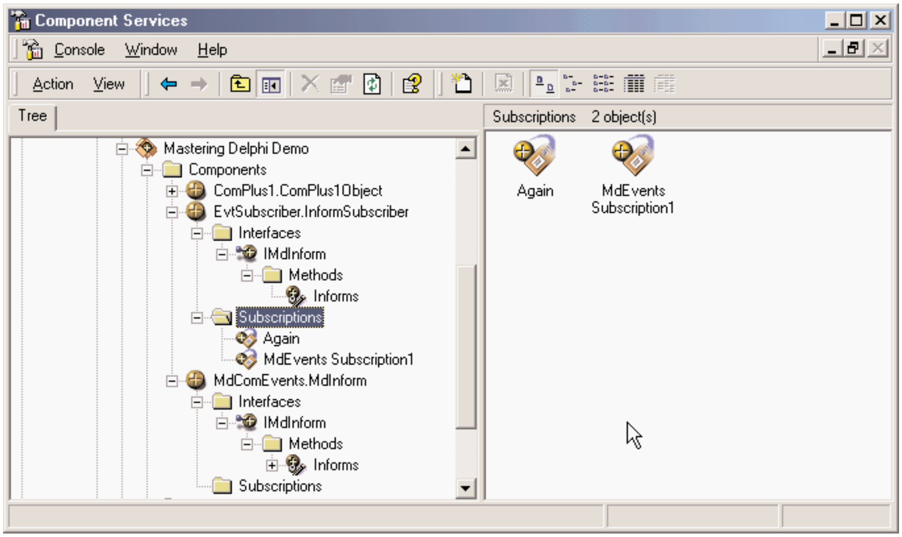


FIGURE 20.26: A COM+ Event with two subscriptions in the Component Services management console.

Finally, we can focus on the application that fires the event, which I've called Publisher, as it publishes the information other COM objects are interested in. This is actually the simplest step of this process, as it is a plain COM client that uses the event server. After importing the COM+ event type library, you can add to the publisher code like this:

```
var
  Inform: IMdInform;
begin
  Inform := CoMdInform.Create;
  Inform.Informs (20, Edit1.Text);
```

My example actually creates the COM object in the `FormCreate` method to keep the reference around, but the effect is the same. Now the client program thinks it is calling the COM+ event object, but this object, provided by the COM+ environment, actually calls the method for each of the active subscribers. In this case you'll end up seeing a message box. To make things a little more interesting, you can actually subscribe twice the same server to the event interface. The net effect is, without touching your client code, you'll get two message boxes, one for each of the subscribed servers.

Obviously this effect becomes interesting when you have multiple different COM components that can handle the event, as you can easily enable and disable each of them in the management console, changing the COM+ environment without modifying the code of any program.

What’s Next?

In this chapter, I have discussed applications of Microsoft’s COM technology, covering automation, documents, controls, and more. We’ve seen how Delphi makes the development of Automation servers and clients, and ActiveX controls, reasonably simple. Delphi even enables us to wrap components around Automation servers, such as Word and Excel.

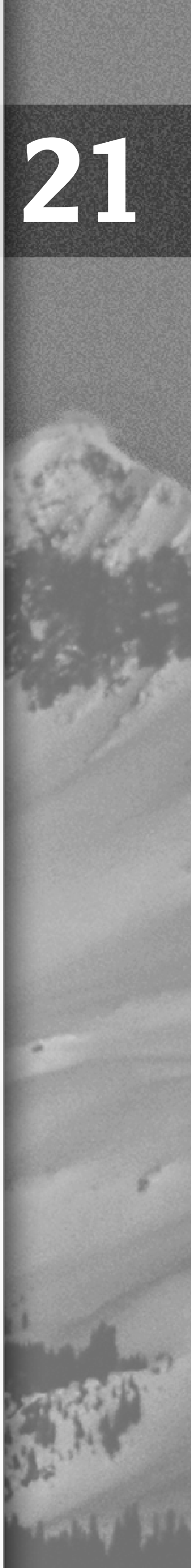
I’ve also introduced elements of COM+ provided by Delphi 6 and discussed briefly the use of ActiveForms inside a browser. I’ve stated this is not really a very good approach to Internet Web programming—the topic discussed in the next two chapters.

As I mentioned earlier, if COM has a key role in Windows 2000, future versions of Microsoft’s operating systems will downplay its role to push the dotNet infrastructure including SOAP and XML. But you’ll have to *wait* until Chapter 23 to see a complete discussion of Delphi 6 XML support.

CHAPTER 21

Internet Programming: Sockets and Indy Components

- Using sockets
- The WinInet API
- Standard Internet actions
- The Internet Direct (Indy) components
- Mail and HTTP



In this chapter I'll provide an introduction to Internet programming in Delphi, using some of the components available in the IDE. With the advent of the Internet era, writing programs based on Internet protocols has become commonplace, so I've devoted three chapters to this topic. This chapter focuses on low-level socket programming and Internet protocols; the next chapter is devoted to server-side Web programming; and the final chapter of the book covers Web services, XML, and SOAP.

We'll start by looking at the use of Delphi socket components, then we'll move to the use of the Internet Direct (Indy) components supporting the most common Internet protocols. I will introduce some elements of HTTP programming, leading up to building HTML files out of database data.

Although you probably just want to use a high-level protocol, our discussion of Internet programming starts from the core concepts and low-level applications. The reason is that understanding TCP/IP and sockets will help you grasp most of the other concepts more easily.

Specifically, I'm going to focus on the use of the connectivity provided by Delphi socket components, which are based on TCP/IP and the low-level Windows sockets. Before we look into the foundations of sockets, let me list a couple of alternative approaches you can use for Internet programming, which I'll cover in more detail in later sections:

- The Delphi socket components provide a good interface for direct use of the Windows sockets API, implementing some custom protocols of your own.
- For standard protocols, you can also use the Indy components, included in Delphi 6.

Foundations of Socket Programming

To understand the description of the socket components in the Delphi Help file, and also to read along with the description of the examples in the book, you need to be confident with several terms related to the Internet in general and with sockets in particular.

The heart of the Internet is the Transmission Control Protocol/Internet Protocol (TCP/IP for short), a combination of two separate protocols that work together to provide connection over the Internet (and can also provide connection over a private intranet). In brief, IP is responsible for defining and routing the *datagrams* (Internet transmission units) and specifying the addressing scheme. TCP is responsible for higher-level transport services.

Configuring a Local Network: IP Addresses

If you have a local network available, you'll be able to test the following programs on it; otherwise, you can simply use the same computer as client and server. In this case, as I've done in the examples, use the address 127.0.0.1 (or *localhost*), which is invariably the address of the current computer. If your network is complex, ask your network administrator to set up proper IP addresses for you. If you want to set up a simple network with a couple of spare computers, you can simply set up the IP address yourself, a 32-bit number usually represented with each of its four components (called *octets*) separated by dots. These numbers have a complex logic underneath them, with the first octet indicating the class of the address.

Specific IP addresses are actually reserved for unregistered internal networks. Internet routers will ignore these address ranges, so you can freely do your tests without interfering with an actual network. For example, the "free" IP address range 192.168.0.0 through 192.168.0.255 can be used for experiments on a network of fewer than 255 machines.

Local Domain Names

How does the IP address map to a name? On the Internet, the client program looks up the values on a domain name server. But it is also possible to have a local *hosts* file, a text file that you can easily edit to provide nice local mappings. You can take a look at the `HOSTS.SAM` file (installed in a subdirectory of the `Windows` directory) to see a sample and then eventually rename the file as `HOSTS`, without the extension, to activate local host mapping.

Should you use an IP or a hostname in your programs? Hostnames are easier to remember and won't require a change if the IP address changes (for whatever reason). On the other hand, IP addresses don't require any resolution, while hostnames must be resolved (a time-consuming operation if the lookup takes place on the Web).

TCP Ports

Each TCP connection takes place though a *port*, which is represented by a 16-bit number. The IP address and the TCP port together specify an Internet connection, or a *socket* (to use a more precise term). Different processes running on the same machine cannot use the same socket—the same port.

Some TCP ports have a standard usage for specific high-level protocols and services. In other words, you should use those port numbers when implementing those services and stay away from them in any other case. Here is a short list:

Protocol	**Port**
HTTP (Hypertext Transfer Protocol)	80
FTP (File Transfer Protocol)	21
SMTP (Simple Mail Transfer Protocol)	25
POP3 (Post Office Protocol, version 3)	110
Telnet	23

The `Services` file (another text file similar to the `Hosts` file) lists the standard ports used by services. You can add your own entry to the list, giving your service a name of your own choosing. Client sockets always specify the port number or the service name of the server socket to which they want to connect.

High-Level Protocols

I've used the term *protocol* many times now, but what does it mean exactly? A protocol is a set of rules the client and server agree upon to determine the communication flow. The low-level Internet protocols, such as TCP/IP, are usually implemented by an operating system. But the term *protocol* is also used for high-level Internet standard protocols (such as HTTP, FTP, or SMTP). These protocols are defined in standard documents available on the Web on the site of the Internet Engineering Task Force (`www.ietf.org`).

If you want to implement a custom communication, you can define your own (possibly simple) protocol, a set of rules determining which request the client can send to the server and how the server can respond to the various possible requests. We'll see an example of a custom protocol later on. Transfer protocols are at a higher level than transmission protocols, because they abstract from the transport mechanism provided by TCP/IP. This makes the protocols independent not only from the operating system and the hardware but also from the physical network.

Socket Connections

How do you start communication through a socket? The server program starts running first, but it simply waits for a request from a client. The client program requests a connection indicating the server it wishes to connect to. When the client sends the request, the server can accept the connection, starting a specific server-side socket, which connects to the client-side socket.

To support this model, there are three different types of socket connections:

- *Client connections* are initiated by the client and connect a local client socket with a remote server socket. Client sockets must describe the server they want to connect to, by providing either its hostname or IP address and its port.
- *Listening connections* are passive server sockets waiting for a client. Once a client makes a new request, the server spawns a new socket devoted to that specific connection and then gets back to listening. Listening server sockets must indicate the port that represents the service they provide. (In fact, the client is going to connect through that port.)
- *Server connections* are the connections activated by servers, as they accept a request from a client.

These different types of connections are important only for establishing the link from the client to the server. Once the link is established, both sides are free to make requests and to send data to the other side.

Delphi Socket Components

Delphi 6 ships with three sets of socket components you can use to read and write information over a TCP/IP connection. The Internet page of the palette hosts the ClientSocket and ServerSocket components (already available in Delphi 5) plus the new TcpClient and TcpServer components (also available in Kylix). To these *native* Borland sockets, the Indy components add the IdTCPClient and the IdTCPServer components. These three sets of components have very similar features, which depend on the underlying protocol. There are technical differences, of course, and platform issues, which can determine your choice.

Host and Port

To use a socket component, you must provide a host and a service. On the server side the host is the address of the current computer; on the client side you can indicate either a domain name or an IP address. The ClientSocket component uses two different properties for these settings (`Host` and `Address`), while the TcpClient component and the IdTCPClient component use a single string property and can determine whether it is a hostname or an address by looking at its content (the property is called `RemoteHost` and `Host`, respectively).

Similarly the service is indicated with the `Port` property or the `Service` property, in a ClientSocket, and with the single `RemotePort` string in a TcpClient and with the single numeric `Port` in an IdTCPClient. The respective servers determine their listening port using analogous properties (called `Port`, `LocalPort`, and `DefaultPort` in the three components).

NOTE The Indy server sockets allow binding to multiple IP addresses and/or ports, using the `Bindings` collection.

Blocking, Nonblocking, and Multithreaded Connections

When working with sockets in Windows, multiple approaches are possible. Reading data from a socket or writing to it can happen asynchronously, so that it does not block the execution of other code in your network application. This is called a *nonblocking connection*. Nonblocking connections read and write asynchronously: the Windows socket support basically sends a message when data is available. Using the ClientSocket and ServerSocket components, for example, the system fires the `OnRead` or `OnWrite` events of the client, and the

OnClientRead or OnClientWrite events of servers inform your socket when the other end of the connection tries to read or write some data.

As an alternative to the asynchronous approach, you can also use *blocking connections*, where your application waits for the reading or writing to be completed before executing the next line of code. In this case, you have to write the code in sequence on both sides, because otherwise the events won't be triggered. When using a blocking connection, you must use a thread on the server, and you'll generally use a thread also on the client.

The Indy components use an in-between approach. They use blocking connections exclusively, and you can either place their code in a thread or use a special helper component (IdAntiFreeze). Using blocking connections for implementing a protocol has the advantage of simplifying the program logic, because you don't have to use the state-machine approach of nonblocking connections, as exemplified later.

Finally, when writing threaded code with the ServerSocket components working on a blocking connection, you can use the TWinSocketStream class to do the actual reading and writing operations. You can use the WaitForData method of the TWinSocketStream class to wait until the socket on the other end is ready to write. You can also create the socket stream class and specify a timeout value, so that if the connection is lost, it won't hang forever.

Using Sockets

After all that theory, let's take a look at a couple of examples. The first is the Sock1 program on the companion CD and is made of the Server1 and Client1 applications, built with the ClientSocket and ServerSocket components in nonblocking mode. The server has a form with the following component:

```
object ServerSocket1: TServerSocket
  Active = True
  Port = 50
  ServerType = stNonBlocking
  OnClientConnect = ServerSocket1ClientConnect
  OnClientDisconnect = ServerSocket1ClientDisconnect
  OnClientRead = ServerSocket1ClientRead
end
```

All the code of the application relates to the events of this component, as the program provides no specific interaction with the user. However, the server has three list boxes for outputting the status, the messages sent from the client, and a log of the events. For example, as a client connects, the server adds the client address to the log:

```
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
```

```
    lbLog.Items.Add ('Connected: ' + Socket.RemoteHost + ' (' +
      Socket.RemoteAddress + ')' );
    PostMessage (Handle, wm_RefreshClients, 0, 0);
  end;
```

Notice that the OnClientConnect event indicates the first occasion for the server to know about the connected client. Using the Socket property, which refers to the low-level TCustomWinSocket, the server can track who is trying to connect. At the end of this and other events, I want to update the list of the connections, using the ActiveConnections property of the server. However, in the OnClientConnect event handler, this list is still not updated, so I post a message to the form to delay the operation:

```
const
  wm_RefreshClients = wm_User;

procedure TForm1.RefreshClients; // message wm_RefreshClients
var
  I: Integer;
begin
  lbClients.Clear;
  for I := 0 to ServerSocket1.Socket.ActiveConnections - 1 do
    with ServerSocket1.Socket.Connections [I] do
      lbClients.Items.Add (RemoteAddress + ' (' + RemoteHost + ')');
end;
```

Similar code is executed as the client disconnects from the server:

```
procedure TForm1.ServerSocket1ClientDisconnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  lbLog.Items.Add ('Disconnected: ' + Socket.RemoteHost + ' (' +
    Socket.RemoteAddress + ')' );
  PostMessage (Handle, wm_RefreshClients, 0, 0);
end;
```

Finally, as the client sends some information to the server (writes to the socket), the server can read the message by calling the ReceiveText function. You should do this read operation only when there is some data available—that is, when the OnClientRead event is fired. Notice also that this is a *destructive* read: the information extracted from the stream is removed from it. Here is the code:

```
procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  // read from the client
```

```
    lbMsg.Items.Add (Socket.RemoteHost + ': ' + Socket.ReceiveText);
  end;
```

Now we can move to the client side of the application, which has a form hosting a client-socket component with the following properties:

```
object ClientSocket1: TClientSocket
  Active = False
  Address = '127.0.0.1'
  ClientType = ctNonBlocking
  Port = 50
  OnConnect = ClientSocket1Connect
  OnDisconnect = ClientSocket1Disconnect
end
```

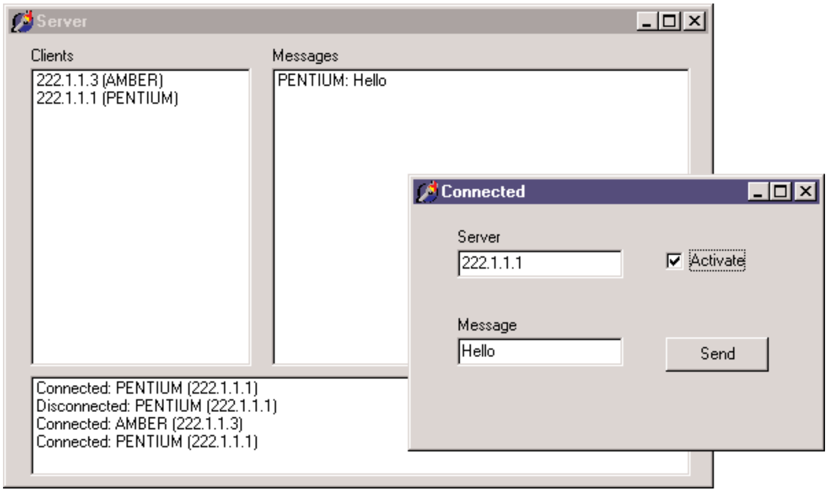
The client form is more interactive. It has two edit boxes and a check box. In the first edit box, you can type the address of the server you want to connect to (to replace the default value listed above), using the check box to activate or deactivate the socket connection:

```
procedure TForm1.cbActivateClick(Sender: TObject);
begin
  if not ClientSocket1.Active then
    ClientSocket1.Address := EditServer.Text;
  ClientSocket1.Active := cbActivate.Checked;
end;
```

As you connect or disconnect, the program simply updates the caption of the form. In the second edit box, you can type a message to send to the server and a button you can press to send the message:

```
procedure TForm1.btnSendClick(Sender: TObject);
begin
  ClientSocket1.Socket.SendText (EditMsg.Text);
end;
```

Notice that this example program doesn't check whether the connection is active before using it, which can result in errors. In Figure 21.1, you can see an example of the client and the server. As the server indicates, a second copy of the client application is running on another computer and is connected to it.



Clients:
222.1.1.3 (AMBER)
222.1.1.1 (PENTIUM)

Messages:
PENTIUM: Hello

Connected: PENTIUM (222.1.1.1)
Disconnected: PENTIUM (222.1.1.1)
Connected: AMBER (222.1.1.3)
Connected: PENTIUM (222.1.1.1)

Server: 222.1.1.1

Message: Hello

FIGURE 21.1: The client and server applications of the Sock1 example, demonstrating the use of the socket components

Using Sockets with a Custom Protocol

Unless you want to send and receive only simple text messages, you might want to define some communication rules between the client and the server. A set of communication rules is generally indicated as a protocol. Basically, the server can receive different requests and, depending on the type of request and whether it can be accomplished, reply to the client.

The server program of the Sock2 example accepts four types of requests: the listing of a directory, a bitmap file, a text file, and the execution of a program on the server. When the server sends back a file, its reply should indicate both what it is going to send back and the actual information. The only method modified from the Sock1 example is the `ServerSocket1ClientRead` procedure, which starts by extracting the five initial characters of the text received by the client that host the command:

```
strCommand := Socket.ReceiveText;
lbLog.Items.Add ('Client: ' + Socket.RemoteAddress + ': ' + strCommand);
// extract the file name (all commands have 5 characters)
strFile := Copy (strCommand, 6, Length (strCommand) - 5);
```

The actual code depends on the initial command defined by the protocol (in this case either *EXEC!* to execute a file on the server, *TEXT!* to return a text file, *BITM!* to retrieve a bitmap file, or *LIST!* to return a directory listing). Here is the code for two of these four alternatives:

```
// send back a text file
if Pos ('TEXT!', strCommand) = 1 then
begin
```

```
    if FileExists (strFile) then
    begin
      strFeedback := 'TEXT!';
      Socket.SendText (strFeedback);
      Socket.SendStream (TFileStream.Create (strFile,
        fmOpenRead or fmShareDenyWrite));
    end
    else
    begin
      strFeedback := 'ERROR' + strFile + ' not found';
      Socket.SendText (strFeedback);
    end;
  end
  // send back a directory listing
  else if Pos ('LIST!', strCommand) = 1 then
  begin
    if DirectoryExists (strFile) then
    begin
      strFeedback := 'LIST!';
      Socket.SendText (strFeedback);
      FileListBox1.Directory := strFile;
      Socket.SendText (FileListBox1.Items.Text);
    end
    else
    begin
      strFeedback := 'ERROR' + strFile + ' not found';
      Socket.SendText (strFeedback);
    end;
  end
  else
  begin
    strFeedback := 'ERROR' + 'Undefined command: ' + strCommand;
    Socket.SendText (strFeedback);
  end;
```

For the directory listings, I've used an invisible FileListBox component. For sending back the text file, I've used the SendStream method, creating a new stream on the fly. The advantage is that there is no need to destroy the temporary stream, as the SendStream method becomes the owner of the stream and destroys it when it is done.

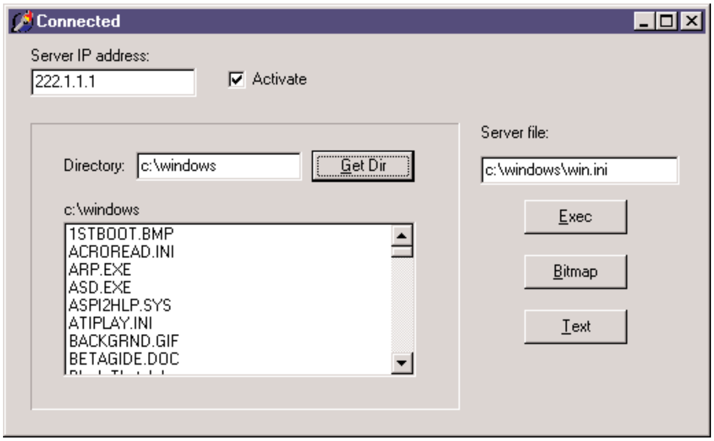
The program sends back multiple pieces of information one after the other. This will create a few problems on the client side, as all the information is received in a single stream. However, the server responds with a five-character header that we can use to determine the content of the rest of the stream. After receiving these headers, the client application sets a status field so that it knows which type of information is coming next. In other words, in the client program, we implement a very simple finite-state machine, a typical technique for socket programming. The client application has five possible states, listed in an enumerated type:

```
type
  TCliStatus = (csIdle, csList, csBitmap, csText, csError);
```

This type is used for the `CliStatus` field of the form. The form has two edit boxes referring to a directory or a file a user can request from the server. When the user presses the Get Dir button, the client program passes to the server the name of the directory indicated by the first edit box. The server will return a list of files, which the client program saves in a list box. At this point, the user can select one of the files from the list box, and the client program will copy it, along with the complete path, into the second edit box. The text of this second edit box is used by the other three buttons—Exec, Bitmap, and Text—which send further requests to the server. In Figure 21.2, you can see an example of the main form of the client program after a directory has been retrieved.

FIGURE 21.2:

The form of the Client2 program after the server has returned the list of the files of a directory



Connected

Server IP address: 222.1.1.1

☑ Activate

Directory: c:\windows

Server file: c:\windows\win.ini

c:\windows

1STBOOT.BMP
ACROREAD.INI
ARP.EXE
ASD.EXE
ASPI2HLP.SYS
ATIPLAY.INI
BACKGRND.GIF
BETAGIDE.DOC

The core of the program is in the `ClientSocket1Read` method, triggered by the socket when there is data to read. The method is first used to get the header indicating which type of data is reaching the program and to set the client program to the proper status:

```
case CliStatus of
  // look for data to receive
  csIdle:
  begin
    Socket.ReceiveBuf (Buffer, 5);
    strIn := Copy (Buffer, 1, 5);
    if strIn = 'TEXT!' then
      CliStatus := csText
    else if strIn = 'BITM!' then
      CliStatus := csBitmap
    // .. and so on
```

Since we don't retrieve all the data, the event is triggered again soon afterward, and this time we are ready to get the actual data. Here are two more branches of the `case` statement:

```
// get a directory listing
csList:
  begin
    ListFiles.Items.Text := Socket.ReceiveText;
    cliStatus := csIdle;
  end;
// read a bitmap file
csBitmap:
  with TFormBmp.Create (Application) do
  begin
    Stream := TMemoryStream.Create;
    Screen.Cursor := crHourglass;
    try
      while True do
      begin
        nReceived := Socket.ReceiveBuf (Buffer, sizeof (Buffer));
        if nReceived <= 0 then
          Break
        else
          Stream.Write (Buffer, nReceived);
        // delay (200 milliseconds)
        Sleep (200);
      end;
      // reset and load the temporary file
      Stream.Position := 0;
      Image1.Picture.Bitmap.LoadFromStream (Stream);
    finally
      Stream.Free;
      Screen.Cursor := crDefault;
    end;
    Show;
    cliStatus := csIdle;
  end;
```

For loading the bitmap, I simply move the data to a `Buffer` (declared as `array [0..9999] of Char`) and then from the buffer to a memory stream, which is later loaded in the Image component of the secondary form. Because the data flow can slow down, the program has a hard-coded delay of 200 milliseconds every time some data is read. Unlike file-reading operations, the loop doesn't stop when the data read is less than the data requested, but only when *no* data is read. (In case of error, the value returned by the `ReceiveBuff` method is –1.)

Sending Database Data over a Socket Connection

Using the techniques we've seen so far, we can write an application that moves database records over a socket. The idea will be to write a front end for data input and a back end for data storage. The client application will have a simple data-entry form and use a database table with string fields for Company, Address, State, Country, Email, and Contact, and a floating-point field for the company ID (called CompID).

NOTE Moving database records over a socket is exactly what you can do with DataSnap and a socket connection component, or with the SOAP support built into Delphi 6 and discussed in Chapter 23, "XML and SOAP."

The client program I've come up with works on a table with this structure saved in the current directory. (You can see the related code in the `OnCreate` event handler.) The core method on the client side is the handler of the `OnClick` event of the Send All button, which sends all the new records to the server. The new records are determined by looking to see whether the record has a valid value for the CompID field. This field, in fact, is not set up by the user but is determined by the server application when the data is sent.

For all new records, the client program packages the field information in a string list, using the structure *FieldName=FieldValue*, obtained using the `Values` property of the string list. The string corresponding to the entire list is then sent to the server. At this point, the program stops in an apparently infinite loop:

```
// save database data in a string list
Data := TStringList.Create;
table1.First;
while not Table1.Eof do
begin
  // if the record is still not logged
  if Table1CompID.IsNull or (Table1CompId.AsInteger = 0) then
  begin
    lbLog.Items.Add ('Sending ' + Table1Company.AsString);
    Data.Clear;
    // create strings with structure "FieldName=Value"
    for I := 0 to Table1.FieldCount - 1 do
      Data.Values [Table1.Fields[I].FieldName] := Table1.Fields [I].AsString;
    // send the record
    ClientSocket1.Socket.SendText (Data.Text);
    // wait for response
    fWaiting := True;
    while fWaiting do
      Application.ProcessMessages;
  end;
  Table1.Next;
end;
```

The program waits forever … or until the handler of another message sets the fWaiting field of the form to False. This happens when the server sends some feedback indicating that the record was received or when the user presses the Stop button. The btnSendAllClick method automatically connects to the server at the beginning and disconnects at the end.

Now let us look at the server. This program has a database table, again stored in the local directory, with two new fields added to the client application's table: LoggedBy, a string field; and LoggedOn, a data field. The values of the two extra fields are determined automatically by the server as it receives data, along with the value of the CompID field. All these operations are done in the ServerSocket1ClientRead method after unpacking the data received by the client:

```
// read from the client
strCommand := Socket.ReceiveText;
// reassemble the data
Data := TStringList.Create;
try
  Data.Text := strCommand;
  // new record
  Table1.Insert;
  // set the fields using the strings
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields [I].AsString := Data.Values [Table1.Fields[I].FieldName];
  // complete with random ID, sender, and date
  Table1CompID.AsInteger := GetTickCount;
  Table1LoggedBy.AsString := Socket.RemoteAddress;
  Table1LoggetOn.AsDateTime := Date;
  Table1.Post;
  // get the value to return
  strFeedback := Table1CompID.AsString;
  // send results back
  lbLog.Items.Add (strFeedback);
  Socket.SendText (strFeedback);
finally
  Data.Free;
end;
```

Except for the fact that some data might be lost, there is no problem when fields have a different order and if they do not match, because the data is stored in the *FieldName=FieldValue* structure. After receiving all the data and posting it to the local table, the server sends back the company ID to the client. The client program, after sending the record, goes into *waiting mode*, a situation modified by receiving feedback from the server:

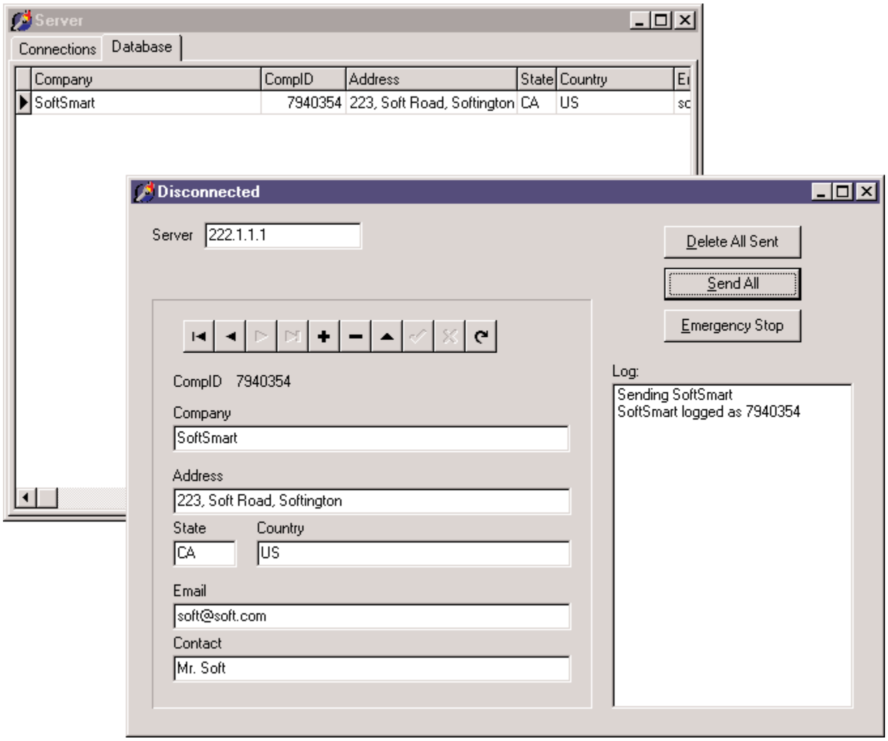
```
procedure TForm1.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  if fWaiting then
  begin
    Table1.Edit;
  Table1CompId.AsString := Socket.ReceiveText;
  Table1.Post;
  lbLog.Items.Add (Table1Company.AsString + ' logged as ' +
   Table1CompId.AsString);
  fWaiting := False;
 end;
end;
```

When receiving feedback, the client program saves the company ID, which marks the record as sent. If the user modifies the record, there is no way to send an update to the server. To accomplish this, you might add a modified field to the client database table and make the server check to see if it is receiving a new field or a modified field. With a modified field, the server should not add a new record but update the existing one.

This is one of the many additions you can make to the program, to make it usable in a real-world environment. The existing code of the program and the previous examples on sockets should provide all you need to complete a similar task. I've limited myself to this version of the application, as shown in Figure 21.3. Notice that the server program has two pages, one with the usual log and the other with a DBGrid showing the current data of the server database table.

FIGURE 21.3:

The client and server programs of the database socket example (DbSock)



Company	CompID	Address	State	Country	E
SoftSmart	7940354	223, Soft Road, Softington	CA	US	sc

Server 222.1.1.1

CompID 7940354

Company SoftSmart

Address 223, Soft Road, Softington

State CA

Country US

Email soft@soft.com

Contact Mr. Soft

Log:

```
Sending SoftSmart
SoftSmart logged as 7940354
```

Working with Blocking Sockets and Threads

A program like the one I've just built is nice, but it won't really scale up on a large system, because the server uses a blocking connection and the requests are processed in sequence. Now, even if making the database-related code multithreading wouldn't be easy, I'll take the excuse of this example to show you how to build a program based on blocking sockets and threads. (If you don't know much about threads, read the sidebar "Working with Threads" before proceeding.)

Working with Threads

Win32 has an API to allow two procedures or methods execute at the same time. Delphi provides a `TThread` class that will let us create and control threads. The first thing to know about the `TThread` class is that you never use it directly, because it is an abstract class—a class with a virtual abstract method. To use threads, you always subclass `TThread` (optionally starting with the Thread Object of the New Items dialog box (File ➢ New ➢ Other) and use the features of this base class.

The `TThread` class has a constructor with a single parameter (`CreateSuspended`) that lets you choose whether to start the thread immediately or suspend it until later. There are also some public synchronization methods:

```
procedure Resume;
procedure Suspend;
function Terminate: Integer;
function WaitFor: Integer;
```

The published properties include `Priority`, `Suspended`, and two read-only, low-level values: `Handle` and `ThreadID`. The class also provides a protected interface, which includes two key methods for your thread subclasses:

```
procedure Execute; virtual; abstract;
procedure Synchronize(Method: TThreadMethod);
```

The `Execute` method, declared as a virtual abstract procedure, must be redefined by each thread class. It contains the main code of the thread, the code you would typically place in a *thread function* when using the Windows API.

The `Synchronize` method is used to avoid concurrent access to VCL components. The VCL code runs inside the main thread of the program, and you need to synchronize access to VCL to avoid reentry problems (errors from reentering a function before a previous call is completed) and concurrent access to shared resources. The only parameter of `Synchronize` is a method that accepts no parameters, typically a method of the same thread class. As you cannot pass parameters to this method, it is common to save some values within the data of the thread object in the `Execute` method and use those values in the *synchronized* methods.

Continued on next page

Another way to avoid conflicts is to use the synchronization techniques offered by the operating system. The SyncObjs unit defines VCL classes for some of these low-level synchronization objects: events (with the `TEvent` class and the `TSingleEvent` class) and critical sections (with the `TCriticalSection` class).

In the server program (see the SockDbThread example on the companion CD), the socket component now has a thread-blocking type and has handlers for many events, including in particular `OnGetThread`. No methods are hooked to the read and write events, as they won't be triggered anymore (they are used exclusively by message-based nonblocking sockets):

```
object ServerSocket1: TServerSocket
  Active = True
  Port = 51
  ServerType = stThreadBlocking
  OnAccept = ServerSocket1Accept
  OnGetThread = ServerSocket1GetThread
  OnClientConnect = ServerSocket1ClientConnect
  OnClientDisconnect = ServerSocket1ClientDisconnect
end
```

The `OnAccept`, `OnClientConnect`, and `OnClientDisconnect` event handlers are used only for logging information to the screen, while the `OnGetThread` event handler has a key role of creating the server-side thread object:

```
procedure TForm1.ServerSocket1GetThread(Sender: TObject; ClientSocket:
  TServerClientWinSocket; var SocketThread: TServerClientThread);
begin
  lbLog.Items.Add('GetThread: ' + ClientSocket.RemoteHost + ' (' +
    ClientSocket.RemoteAddress + ')' );
  SocketThread := TDbServerThread.Create(False, ClientSocket);
end;
```

This must be an object of a class inherited by the specific `TServerClientThread` class (not the generic Delphi `TThread` class) and with its core code placed in an overridden `ClientExecute` method (not the generic `Execute` method):

```
type
  TDbServerThread = class(TServerClientThread)
  private
    strCommand: string;
    strFeedback: string;
  public
    procedure ClientExecute; override;
    procedure Log;
    procedure LogFeedback;
    procedure AddRecord;
  end;

procedure TDbServerThread.ClientExecute;
var
  Stream: TWinSocketStream;
  Buffer, strIn: string;
  nRead: Integer;
begin
  // keep going
  Stream := TWinSocketStream.Create(ClientSocket, 5000);
  try
    while not Terminated and ClientSocket.Connected do
    begin
      // initialize (thread might be reused)
      Buffer := '';
      strIn := '';
      SetLength(Buffer, 64);
      repeat
        nRead := Stream.Read(Buffer[1], 64);
        if nRead = 0 then
        begin
          ClientSocket.Close;
          Break;
        end;
        SetLength (Buffer, nRead);
        StrIn := StrIn + Buffer;
      until (Pos(#10#13'.'#10#13, Buffer) > 0);

      if strIn = '' then
        Continue // keep going
      else
      begin
        // handle the request, if anything arrived
        StrCommand := Copy (strIn, 1, Pos (#10#13'.'#10#13, strIn) -1);
        Synchronize(Log);
        Synchronize(AddRecord);
        // send results back
        Synchronize(LogFeedback);
        Stream.Write(strFeedback[1], Length (strFeedback));
      end;
    end;
  finally
    Stream.Free;
  end;
end;
```

The server reads data from the client and sends the feedback using a TWinSocketStream, a compulsory approach for blocking servers. The thread is kept active, as it can be reused for subsequent calls, and reads from a buffer until it find a specific separator (in this case, a dot between two line separators, exactly as occurs in the SMTP protocol). When the data is received, the server does a *synchronized* call to the AddRecord method, which is similar to the code of the previous version of the example.

WARNING This approach is far from perfect, as all database accesses are serialized, but you could solve the problem by moving the database access component within the thread and adding a Session object in case of a BDE application. Not all databases like multithreaded access, though, so serializing the calls is not always a bad idea.

This is the multithreaded server application. The client program, instead, uses a standard thread class, derived from TThread. The class creates a client server object internally, so that multiple threads could spawn multiple socket connections in parallel (something the program doesn't really use) and receives a table to work on as parameter in the constructor:

```
type
  TLogEvent = procedure(Sender: TObject; LogMsg: String) of object;

  TSendThread = class(TThread)
  private
    ClientSocket: TClientSocket;
    FTable: TTable;
    FOnLog: TLogEvent;
    FLogMsg: String;
    procedure SetOnLog(const Value: TLogEvent);
  protected
    procedure Execute; override;
    procedure DoLog;
  public
    constructor Create(ATable: TTable);
    property OnLog: TLogEvent read FOnLog write SetOnLog;
  end;
```

On the client side, the finite-state machine logic (send, set the wait flag, receive another event, disable the flag, get the next record) is now replaced by a continual and more logical flow of operations, all part of the Execute method of the thread. You need to add some code to let the program wait until the server sends a reply. Again, the blocking socket of the client is not used directly but via a TWinSocketStream object:

```
procedure TSendThread.Execute;
var
  I: Integer;
  Data: TStringList;
  Stream: TWinSocketStream;
  Buf: String;
begin
  try
    Data := TStringList.Create;
    ClientSocket := TClientSocket.Create (nil);
    Stream := nil;
    try
      ClientSocket.Address := EditServer.Text;
      ClientSocket.ClientType := ctBlocking;
      ClientSocket.Port := 51;
      ClientSocket.Active := True;
      Stream := TWinSocketStream.Create(ClientSocket.Socket, 30000);

      FTable.First;
      while not FTable.Eof do
      begin
        // if the record is still not logged
        if FTable.FieldByName('CompID').IsNull or
          (FTable.FieldByName('CompID').AsInteger = 0) then
        begin
          FLogMsg := 'Sending ' + Table.FieldByName('Company').AsString;
          Synchronize(DoLog);
          Data.Clear;
          // create strings with structure "FieldName=Value"
          for I := 0 to FTable.FieldCount - 1 do
            Data.Values [FTable.Fields[I].FieldName] :=
              FTable.Fields [I].AsString;
          // send the record followed by separator
          Buf := Data.Text + #10#13'.'#10#13;
          ClientSocket.Socket.SendText(Buf);
          // wait for reponse
          if Stream.WaitForData(30000) then
          begin
            FTable.Edit;
            SetLength(Buf, 256);
            SetLength(Buf, Stream.Read(Buf[1], Length(Buf)));
            FTable.FieldByName('CompID').AsString := Buf;
            FTable.Post;
            FLogMsg := FTable.FieldByName('Company').AsString +
              ' logged as ' + FTable.FieldByName('CompID').AsString;
          end
          else
            FlogMsg := 'No response for ' +
              FTable.FieldByName('Company').AsString;
          Synchronize(DoLog);
        end;
        FTable.Next;
      end;
    finally
      ClientSocket.Active := False;
      ClientSocket.Free;
      Stream.Free;
      Data.Free;
    end;
  except
    // trap exceptions
  end;
end;
```

The thread also has an event handler to let the forms using it define the effect of the `OnLog` operation, in the synchronized `DoLog` method. Finally, this is how the thread starts, when the user clicks a button in the form:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  SendThread: TSendThread;
begin
  SendThread := TSendThread.Create(Table1);
  SendThread.OnLog := OnLog;
  SendThread.Resume;
end;
```

Internet Protocols

After discussing the low-level socket components, we are ready to delve into the core topic of this chapter, the use of higher-level Internet protocols.

As already mentioned, Delphi 6 now ships with a collection of open-source Internet components called Internet Direct, or Indy for short. The Indy components, previously called WinShoes (a pun on the term WinSockets), are built by a group of developers led by Chad Hower and are available also in Kylix. You can find more information and possibly updated versions of the actual components at `www.nevrona.com/indy`.

The Indy components are available within the Delphi IDE, but they are not the only set of Internet components. Delphi 6 Component Palette also has another page of Internet protocol components, the FastNet page. These are available for compatibility with Delphi 5 and earlier versions. Third-party solutions, both freely available or for sale, also provide implementations of Internet protocols.

Here and in the next chapter, I'm going to focus exclusively on the Indy components. This chapter focuses on the use of Internet protocols within Windows applications, while the next shows examples of the use of these protocols within Web server applications.

Sending and Receiving Mail

Probably the most common operation you do on the Internet is to send and receive e-mail. There is generally very little need to write a complete application to handle e-mail, as some of the existing programs are actually rather complete. For this reason, I have no intention of writing a general-purpose mail program here. You can find some examples of those among Delphi Internet demos.

Other than creating a general-purpose mail application, what else can one do with the mail components and protocols? There are many possibilities, which I've tried to group in two areas:

Automatic generation of mail messages An application you've written can have an About box for sending a registration message back to your marketing department or a specific menu item for sending a request to your tech support. You might even decide to enable a tech-support connection whenever an exception occurs. Another related task would be automating the dispatching of a message to a list of people or generating an automatic message from your Web site, an example I'll show you toward the end of this chapter.

Use of mail protocols for communication with users who are only occasionally online When you must move data between users who are not always online, you can write an application on a server to synchronize among them, and you can give each user a specialized client application for interacting with the server. An alternative is to use an existing server application, such as a mail server, and write the two specialized programs based on the mail protocols. The data sent over this connection will generally be formatted in special ways, so you'll want to use a specific e-mail addresses for these messages (not your primary e-mail address). As an example, you could rewrite the earlier DbSock example to dispatch mail messages instead of using a custom socket connection. This will give you the advantage of being firewall-friendly and allowing the server to be temporarily offline, as the requests would be kept on the mail server.

Sending Messages with Your Mail Program

The simplest technique for automating the generation of an e-mail message is to use your existing mail application, adding a message to its outbox. Using the `ShellExecute` API function, you can easily send a message to the default mail program registered on the computer.

To test this technique, I've prepared a simple form with two edit boxes and a memo for the input. Pressing a button creates a string with all the information about the message and then sends it, simply executing the string with the *mailto:* prefix. Here is the code of the Send button of the MailGen example from the companion CD:

```
uses
  ShellApi;

procedure TForm1.BtnSendClick(Sender: TObject);
var
  strMsg: string;
  I: Integer;
begin
  // set the basic information
  strMsg := 'mailto:' + EditAddress.Text + '?Subject=' + EditSubject.Text +
    '&Body=';
  // add first line
  if Memo1.Lines.Count > 0 then
    strMsg := strMsg + Memo1.Lines [0];
  // add subsequent lines separated by the newline symbol
  for I := 1 to Memo1.Lines.Count - 1 do
    strMsg := strMsg + '%0D%0A' + Memo1.Lines [I];
  // send the message
  ShellExecute (Handle, 'open', pChar (strMsg), '', '', SW_SHOW);
end;
```

To show the body of the message on multiple lines, you can separate each line with the carriage return and line feed characters (usually indicated in Delphi as #13 and #10). These values should be explicitly added to the string in hexadecimal format and prefixed by the % sign, as required by a URL. You can actually obtain this encoding automatically by using the NMURL component.

NOTE You can also send mail with the `TSendMail` predefined action, which is based on the MAPI standard.

Mail In and Out

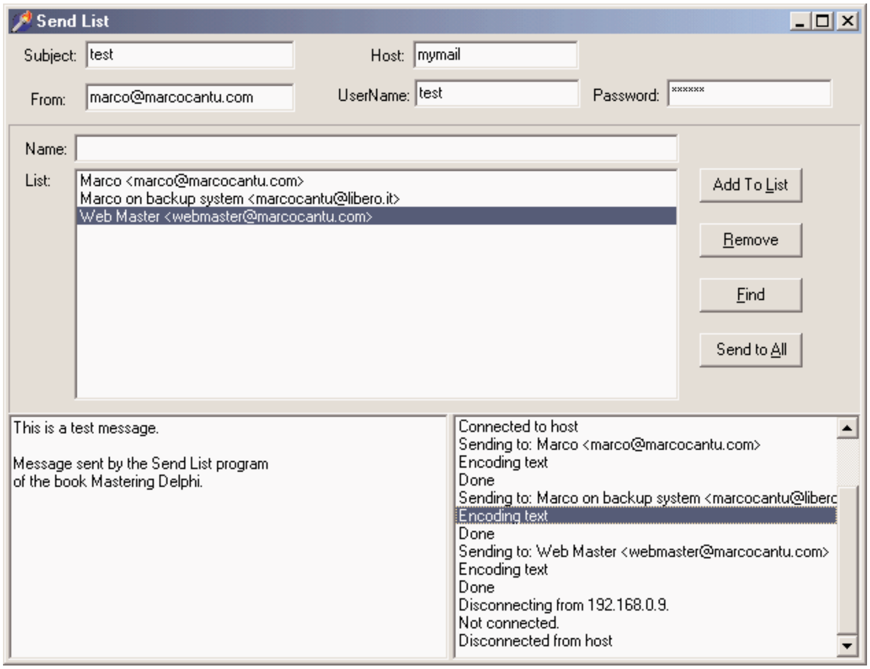
To showcase the development of simple e-mail management programs, I could build an example of how you can send and receive mail. The Indy components, though, include a rather complete set of examples, and I don't see any reason to duplicate those, as using the mail protocols means placing a message component (IdMessage) in your application, filling it with data, and then using the IdSMTP component to send the mail message. To *retrieve* a mail message from your mailbox, use the IdPop3 component, which will return you an IdMessage object.

Just to give you an idea of how this works, I've written a program for sending mail to multiple people at once, using a list stored in an ASCII file. I originally used this program myself for sending mail to people who sign up on my Web site, but later I extended the program by adding database support and reading subscriber logs automatically. The original version of the program is still a good introduction to the use of the SMTP component of Indy.

The SendList program keeps a list of names and e-mail addresses in a local file, which is displayed in a list box. A few buttons allow you to add and remove items, or modify them by removing the item, editing it, and then adding the item again. When the program closes, the updated list is automatically saved. Now let's get to the interesting portion of the program. The top-most panel, shown in Figure 21.4, allows you to enter the subject, the sender address, and the information used to connect to the mail server (hostname, username, and eventually a password).

FIGURE 21.4:

The SendList program in action



Subject: test
From: marco@marcocantu.com
Host: mymail
UserName: test
Password: ******
Name:
List:
Marco <marco@marcocantu.com>
Marco on backup system <marcocantu@libero.it>
Web Master <webmaster@marcocantu.com>

This is a test message.

Message sent by the Send List program
of the book Mastering Delphi.

```
Connected to host
Sending to: Marco <marco@marcocantu.com>
Encoding text
Done
Sending to: Marco on backup system <marcocantu@liberc
Encoding text
Done
Sending to: Web Master <webmaster@marcocantu.com>
Encoding text
Done
Disconnecting from 192.168.0.9.
Not connected.
Disconnected from host
```

You'll probably want to make the value of these edit boxes persistent, possibly in an INI file. I haven't done this, only because I don't really want you to see my mail connection details! The value of these edit boxes, along with the list of addressee, allows you to send the series of mail messages, after customizing each of them, with the following code:

```
procedure TMainForm.BtnSendAllClick(Sender: TObject);
var
  nItem: Integer;
  Res: Word;
begin
  Res := MessageDlg ('Start sending from item ' +
    IntToStr (ListAddr.ItemIndex) + ' (' +
    ListAddr.Items [ListAddr.ItemIndex] + ')?'#13 +
    '(No starts from 0)', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
  if Res = mrCancel then
    Exit;
  if Res = mrYes then
    nItem := ListAddr.ItemIndex
  else
    nItem := 0;
  // connect
  Mail.Host := eServer.Text;
  Mail.UserID := eUserName.Text;
  if ePassword.Text <> '' then
  begin
    Mail.Password := ePassword.Text;
    Mail.AuthenticationType := atLogin;
  end;
  Mail.Connect;
  // send the messages, one by one, prepending a custom message
  try
    // set the fixed part of the header
    MailMessage.From.Name := eFrom.Text;
    MailMessage.Subject := eSubject.Text;
    MailMessage.Body.SetText (reMessageText.Lines.GetText);
    MailMessage.Body.Insert (0, 'Hello');
    while nItem < ListAddr.Items.Count do
    begin
      // show the current selection
      Application.ProcessMessages;
      ListAddr.ItemIndex := nItem;
      MailMessage.Body [0] := 'Hello ' + ListAddr.Items [nItem];
      MailMessage.Recipients.EMailAddresses := ListAddr.Items [nItem];
      Mail.Send(MailMessage);
      Inc (nItem);
    end;
  finally // we're done
    Mail.Disconnect;
  end;
end;
```

Another interesting example of the use of the mail is to notify developers of problems within applications, something you might want to use more in an internal application than in one you'll distribute widely. You can obtain this effect by modifying the ErrorLog example of Chapter 4, "The Run-Time Library," and sending mail when an exception (or one of a given type only) occurs.

Working with HTTP

Handling mail messages is certainly interesting, and mail protocols are probably still the most widespread Internet protocols. The other popular protocol is HTTP, the one used by Web servers and Web browsers. This is the protocol to which we'll devote the rest of this chapter and all of the following.

On the client side of the Web, the main activity is browsing—reading HTML files. Besides building a custom browser, you can embed the Internet Express ActiveX control within your program (as I've done in WebDemo example in Chapter 20, "From Automation to COM+"). You can also directly activate the browser installed on the computer of the user, for example, opening an HTML page by calling the `ShellExecute` method (defined in the ShellApi unit):

```
ShellExecute (Handle, 'open', FileName, '', '', sw_ShowNormal);
```

Using `ShellExecute`, we can simply execute a document, such as a file. Windows will start the program associated with the HTM extension, using the action passed as the parameter (in this case, *open*). You can use a similar call to view a Web site, by using a string like '*http://www.example.com*' instead of a filename. In this case, the system recognizes the *http* section of the request as requiring a Web browser and launches it.

On the server side, you generate and make available the HTML pages. At times, it may be enough to have a way to produce static pages, occasionally extracting new data from a database table to update the HTML files as needed. In other cases, you'll need to generate pages dynamically based on a request from a user.

As a starting point, I'll discuss HTTP by building a simple but complete client and server, then we'll move on to discussing HTML producer components and introducing the Web server extension technologies (CGI and ISAPI). In the next chapter, we'll move from this "core technology" level to the RAD development style for the Web supported by Delphi 6, discussing the WebBroker and WebSnap architectures.

Grabbing HTTP Content

As an example of the use of the HTTP protocols, I've decided to write a very specific search application. The program simply hooks onto the Google Web site, searches for a keyword, and retrieves the first hundred sites found. Instead of showing the resulting HTML file, the program parses it to extract only the URLs of the related sites to a list box. The description of these sites is kept in a separate string list and is displayed as you click a list-box item. So the program demonstrates two techniques at once: retrieving a Web page and parsing its HTML code.

To demonstrate how you should work with blocking connections, such as those used by Indy, I've implemented the program using a background thread for the actual processing. (See the sidebar "Working with Threads," earlier in this chapter, for a very short introduction to this topic.) This approach also gives the advantage of being able to start multiple searches at once. The thread class used by the WebFind application receives as input a URL to look for, `strUrl`.

The class has two output procedures, `AddToList` and `ShowStatus`, to be called inside the `Synchronize` method. The code of these two methods sends some results or some feedback to the main form, respectively adding a line to the listbox and changing the status bar `SimpleText` property. The key method of the thread is the `Execute` method. Before we look at it, however, let me show you how the thread is activated by the main form:

```
const
  strSearch = 'http://www.google.com/search?as_q=';

procedure TForm1.BtnFindClick(Sender: TObject);
var
  FindThread: TFindWebThread;
begin
  // create suspended, set initial values, and start
  FindThread := TFindWebThread.Create (True);
  FindThread.FreeOnTerminate := True;
  // grab the first 100 entries
  FindThread.strUrl := strSearch + EditSearch.Text +'&num=100';
  FindThread.Resume;
end;
```

The URL string is made of the main address of the search engine, followed by some parameters. The first, `as_q`, indicates the words you are looking for. The second, `num=100`, indicates the number of sites to retrieve; you cannot use numbers at will but are limited to few alternatives, with 100 being the largest possible value.

WARNING The WebFind program works with the server on the Google Web site at the time this book was written and tested. The custom software on the site can change any day, however, which might prevent WebFind from operating correctly.

The Execute method of the thread, activated by the Resume call, simply calls the two methods actually doing the work and shown in Listing 21.1. In the first, GrabHtml, the program connects to the HTTP server using a dynamically created IdHttp component, and reads the HTML with the result of the search. The second method, HtmlToList, extracts the URLs referring to other Web sites from the result, the strRead string.

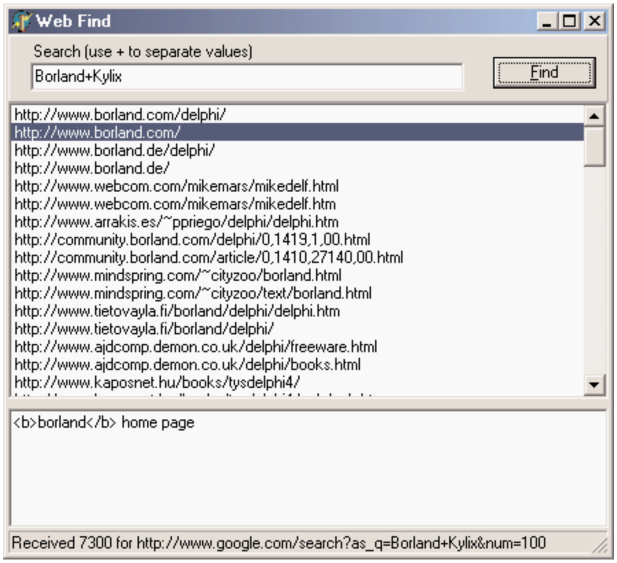
Listing 21.1: The *TFindWebThread* class (of the WebFind program)

```
unit FindTh;

interface

uses
  Classes, IdComponent, SysUtils, IdHTTP;

type
  TFindWebThread = class(TThread)
  protected
    Addr, Text, Status: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (Sender: TObject; AWorkMode: TWorkMode;
      const AWorkCount: Integer);
  public
    strUrl: string;
    strRead: string;
  end;

implementation

{ TFindWebThread }

uses
  WebFindF;

procedure TFindWebThread.AddToList;
begin
  if Form1.ListBox1.Items.IndexOf (Addr) < 0 then
  begin
    Form1.ListBox1.Items.Add (Addr);
    Form1.DetailsList.Add (Text);
  end;
end;

procedure TFindWebThread.Execute;
begin
```

```
  GrabHtml;
  HtmlToList;
  Status := 'Done with ' + StrUrl;
  Synchronize (ShowStatus);
end;

procedure TFindWebThread.GrabHtml;
var
  Http1: TIdHTTP;
begin
  Status := 'Sending query: ' + StrUrl;
  Synchronize (ShowStatus);
  Http1 := TIdHTTP.Create (nil);
  try
    Http1.OnWork := HttpWork;
    strRead := Http1.Get (StrUrl);
  finally
    Http1.Free;
  end;
end;

procedure TFindWebThread.HtmlToList;
var
  strAddr, strText: string;
  nText: integer;
  nBegin, nEnd: Integer;
begin
  Status := 'Elaborating data for: ' + StrUrl;
  Synchronize (ShowStatus);
  strRead := LowerCase (strRead);
  repeat
    // find the initial part HTTP reference
    nBegin := Pos ('href=http', strRead);
    if nBegin <> 0 then
    begin
      // get the remaining part of the string, starting with 'http'
      strRead := Copy (strRead, nBegin + 5, 1000000);
      // find the end of the HTTP reference
      nEnd := Pos ('>', strRead);
      strAddr := Copy (strRead, 1, nEnd - 1);
      // move on
      strRead := Copy (strRead, nEnd + 1, 1000000);
      // add the URL if 'google' is not in it
      if Pos ('google', strAddr) = 0 then
      begin
        nText := Pos ('</a>', strRead);
        strText := copy (strRead, 1, nText - 1);
        // remove cached references and duplicates
        if (Pos ('cached', strText) = 0) then
        begin
          Addr := strAddr;
          Text := strText;
          AddToList;
        end;
      end;
    end;
  until nBegin = 0;
end;

procedure TFindWebThread.HttpWork(Sender: TObject; AWorkMode: TWorkMode;
  const AWorkCount: Integer);
begin
  Status := 'Received ' + IntToStr (AWorkCount) + ' for ' + strUrl;
  Synchronize (ShowStatus);
end;

procedure TFindWebThread.ShowStatus;
begin
  Form1.StatusBar1.SimpleText := Status;
end;

end.
```

The program looks for subsequent occurrences of the `href="http` substring, copying the text up to the closing > character. If the found string contains the word *google*, or its target text includes the word *cached*, it is omitted from the result. You can see the effect of this code in the output of Figure 21.5. Notice that I've already gotten the result of a request, but the program is currently retrieving another page, as indicated in the status bar. You can start multiple searches at the same time, but be aware that the results will all be added to the same memo component.

FIGURE 21.5:
The WebFind application can be used to search for a list of sites on the Google search engine.



Search (use + to separate values)
Borland+Kylix

http://www.borland.com/delphi/
http://www.borland.com/
http://www.borland.de/delphi/
http://www.borland.de/
http://www.webcom.com/mikemars/mikedelf.html
http://www.webcom.com/mikemars/mikedelf.htm
http://www.arrakis.es/~ppriego/delphi/delphi.htm
http://community.borland.com/delphi/0,1419,1,00.html
http://community.borland.com/article/0,1410,27140,00.html
http://www.mindspring.com/~cityzoo/borland.html
http://www.mindspring.com/~cityzoo/text/borland.html
http://www.tietovayla.fi/borland/delphi/delphi.htm
http://www.tietovayla.fi/borland/delphi/
http://www.ajdcomp.demon.co.uk/delphi/freeware.html
http://www.ajdcomp.demon.co.uk/delphi/books.html
http://www.kaposnet.hu/books/tysdelphi4/

<b>borland</b> home page

Received 7300 for http://www.google.com/search?as_q=Borland+Kylix&num=100

The WinInet API

When you need to use the FTP and HTTP protocols, as alternatives to using particular VCL components, you can use a specific API provided by Microsoft in the WinInet DLL. This library is part of the core operating system and implements the FTP and HTTP protocols on top of the Windows sockets API.

With just three calls—`InternetOpen`, `InternetOpenURL`, and `InternetReadFile`—you can retrieve a file corresponding to any URL and store a local copy or analyze it. Other simple methods can be used for FTP; I suggest you look for the source code of the Delphi unit, listing all the functions, and for the specific Help file for the DLL, which is not part of the SDK Help shipping with Delphi.

The `InternetOpen` function establishes a generic connection and returns a handle you can use in the `InternetOpenURL` call. This second call returns a handle to the URL that you can pass to the `InternetReadFile` function in order to read blocks of data. In the following sample code, the data is stored in a local string. When all the data has been read, the program closes the connection to the URL and the Internet session by calling the `InternetCloseHandle` function twice.

```
var
  hHttpSession, hReqUrl: HInternet;
  Buffer: array [0..1023] of Char;
  nRead: Cardinal;
  strRead: string;
  nBegin, nEnd: Integer;
begin
  strRead := '';
  hHttpSession := InternetOpen ('FindWeb', INTERNET_OPEN_TYPE_PRECONFIG,
    nil, nil, 0);
  try
    hReqUrl := InternetOpenURL (hHttpSession, PChar(StrUrl), nil, 0,0,0);
    try // read all the data
      repeat
        InternetReadFile (hReqUrl, @Buffer, sizeof (Buffer), nRead);
        strRead := strRead + string (Buffer);
      until nRead = 0;
    finally
      InternetCloseHandle (hReqUrl);
    end;
  finally
    InternetCloseHandle (hHttpSession);
  end;
end;
```

Browsing on Your Own

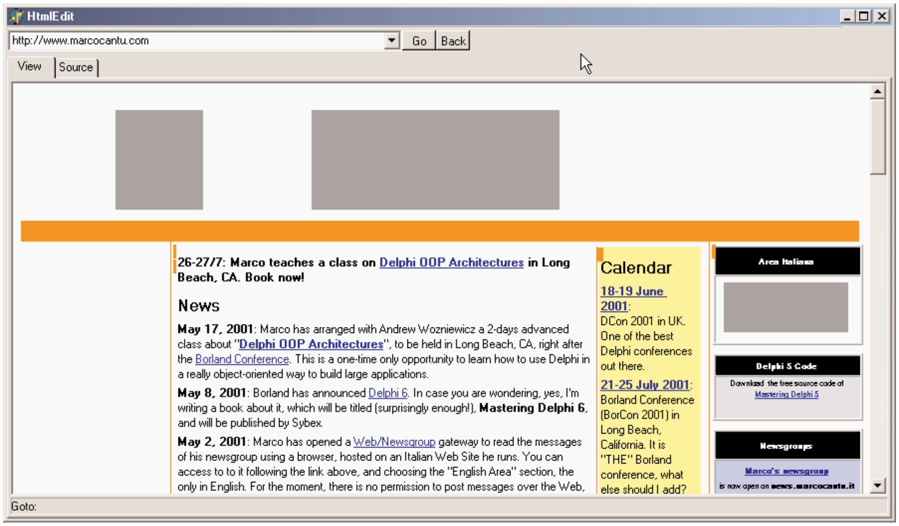
Although I doubt you are interested in writing a new Web browser, it might be interesting anyway to see how you can grab an HTML file from the Internet and display it locally, using the HTML viewer available in CLX (the TextBrowser control). Connecting this control to an Indy HTTP client, you can come up with a simplistic text-only browser with limited navigation in minutes. The core is to write

```
TextBrowser1.Text := IdHttp1.Get (NewUrl);
```

where NewUrl is complete location of the Web resource you want to access to. In the BrowseFast example on the CD-ROM, this URL is entered in a combo box, which keeps track of recent requests. The effect of a similar call is to return the textual portion of a Web page (see Figure 21.6), as grabbing the graphic content requires much more complex coding. The TextBrowser control, in fact, is better defined as a local file viewer than as a browser.

FIGURE 21.6:

The output of the BrowseFast text-only browser



26-27/7: Marco teaches a class on Delphi OOP Architectures in Long Beach, CA. Book now!

News

May 17, 2001: Marco has arranged with Andrew Wozniewicz a 2-days advanced class about "**Delphi OOP Architectures**", to be held in Long Beach, CA, right after the Borland Conference. This is a one-time only opportunity to learn how to use Delphi in a really object-oriented way to build large applications.

May 8, 2001: Borland has announced Delphi 6. In case you are wondering, yes, I'm writing a book about it, which will be titled (surprisingly enough!), **Mastering Delphi 6**, and will be published by Sybex.

May 2, 2001: Marco has opened a Web/Newsgroup gateway to read the messages of his newsgroup using a browser, hosted on an Italian Web Site he runs. You can access to to it following the link above, and choosing the "English Area" section, the only in English. For the moment, there is no permission to post messages over the Web,

Calendar

18-19 June 2001: DCon 2001 in UK. One of the best Delphi conferences out there.

21-25 July 2001: Borland Conference (BorCon 2001) in Long Beach, California. It is "THE" Borland conference, what else should I add?

Area Italiana

Delphi 5 Code

Download the free source code of Mastering Delphi 5

Newsgroups

Marco's newsgroup

is now open on **news.marcocantu.it**

In any case, I've added to the program only very limited support for hyperlinks. When a user moves the mouse over a link, its link text is copied to a local variable (NewRequest), which is then used in case of a click on the control to compute the new HTTP request to forward. Merging the current address (LastUrl) with the request, though, is far from trivial, even with the help of the IdUrl class provided by Indy. Here is the code I've come up with, which handles only the simplest cases:

```
procedure TForm1.TextBrowser1Click(Sender: TObject);
var
  Uri: TIdUri;
```

```
begin
  if NewRequest <> '' then
  begin
    Uri := TIdUri.Create (LastUrl);
    if Pos ('http:', NewRequest) > 0 then
      GoToUrl (NewRequest)
    else if NewRequest [1] = '/' then
      GoToUrl ('http://' + Uri.Host + NewRequest)
    else
      GoToUrl ('http://' + Uri.Host + Uri.Path + NewRequest);
  end;
end;
```

Again, this example is really trivial and is far from usable, but building a browser involves a *little* more than the ability to connect via HTTP and display HTML files.

A Simple HTTP Server

The situation with the development of an HTTP server is quite different. Building a server to deliver static pages based on HTML files is far from simple, although one of the Indy demos provides a rather good starting point for this. A custom HTTP server, instead, might be interesting when building a totally dynamic site, something I'll focus on in more detail in the next chapter.

To show you how you can start the development of a custom HTTP server, I've built the HttpServ example. This program has a form with a list box used for logging requests and an IdHTTPServer component, with these settings:

```
object IdHTTPServer1: TIdHTTPServer
  Active = True
  DefaultPort = 8080
  OnCommandGet = IdHTTPServer1CommandGet
end
```

The server uses the port 8080 instead of the standard port 80, so that you can run it alongside another Web server. All of the custom code is in the `OnCommandGet` event handler, which simply returns a fixed page plus some information about the request itself:

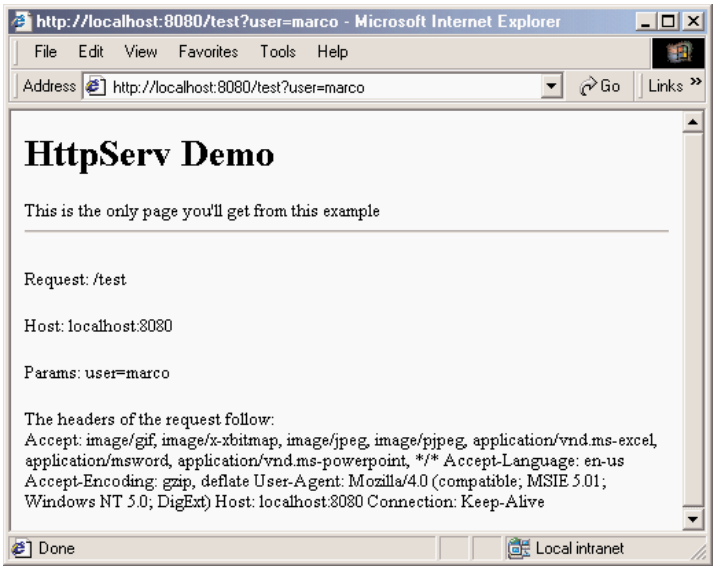
```
procedure TForm1.IdHTTPServer1CommandGet(AThread: TIdPeerThread;
  RequestInfo: TIdHTTPRequestInfo; ResponseInfo: TIdHTTPResponseInfo);
var
  HtmlResult: String;
begin
  // log
  Listbox1.Items.Add (RequestInfo.Document);
  // respond
  HtmlResult := '<h1>HttpServ Demo</h1>' +
      '<p>This is the only page you''ll get from this example.</p><hr>' +
      '<p>Request: ' + RequestInfo.Document + '</p>' +
      '<p>Host: ' + RequestInfo.Host + '</p>' +
      '<p>Params: ' + RequestInfo.UnparsedParams + '</p>' +
      '<p>The headers of the request follow: <br>' +
      RequestInfo.Headers.Text + '</p>';
    ResponseInfo.ContentText := HtmlResult;
  end;
```

By passing a path and some parameters in the command line of the browser, you'll see them reinterpreted and displayed. For example, Figure 21.7 shows the effect of the command line:

```
http://localhost:8080/test?user=marco
```

FIGURE 21.7:

The page displayed by connecting a browser to the custom HttpServ program



http://localhost:8080/test?user=marco - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Address
http://localhost:8080/test?user=marco
Go
Links
Done
Local intranet

HttpServ Demo

This is the only page you'll get from this example

Request: /test

Host: localhost:8080

Params: user=marco

The headers of the request follow:
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */* Accept-Language: en-us Accept-Encoding: gzip, deflate User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0; DigExt) Host: localhost:8080 Connection: Keep-Alive

If this example seems too trivial, you'll see a slightly more interesting version in the next section, as I discuss the generation of HTML with Delphi's producer components.

NOTE If you plan building an advanced Web server or other Internet servers with Delphi, then as an alternative to the Indy components, have a look at the DXSock components from Brain Patchwork DX (`www.dxsock.com`).

Generating HTML

The Hypertext Markup Language, better known by its acronym HTML, is the most widespread format for content on the Web. HTML is the format Web browsers typically read; it is a standard defined by the W3C, the World Wide Web Consortium, which is one of the bodies controlling the Internet. The current standard is represented by HTML 4, although not all browsers fully support that. When building a Web site, you always need to choose a lowest-common-denominator approach to support most of the browsers in use—that is, unless you are targeting a specific group of users whom you ask to adopt a specific browser (as happens in intranet situations). If you don't know much about the tags included in HTML files, you may want to read the sidebar "The Format of HTML Files" for a fast introduction.

The Format of HTML Files

If you have a little familiarity with HTML but don't work with it often enough to have all the basic elements "down cold," here's a quick summary.

HTML files are basically ASCII text files. Besides plain text, an HTML file contains many tags, which might determine the style of the font, the type of paragraph, or a link to another HTML file or an image, among other things.

Most tags are paired as opening tags and closing tags (the closing tag is usually the same as the opening tag but is preceded by a slash, /) to indicate where the style or content begins and ends. For example, you write `<b>important</b>` to set the word *important* in bold, and you write `<title>Document Title</title>` to define the title of a document as *Document Title*. (A few elements, such as `<br>` for a line break and `<img>` for a graphic or "image," stand alone and do not use a matching closing tag.)

An HTML document begins with the `<html>` tag and is divided into two parts, marked as `<head>` and `<body>`. Each of these three tags requires the corresponding terminator. In the head portion of the HTML file, you'll generally write the title (often displayed in the title bar of the browser) and a few other generic elements.

In the body, you write the contents of the file, generally starting with its visible title. You can use headings with different levels, marked with the `<hX>` tag, where you'd replace *X* with a number from 1 to 6. These are followed by plain paragraphs (`<p>`), preformatted paragraphs (`<pre>`, a style generally used for program listings), various types of lists, and many other elements. The text will often have links to other pages or other parts of the current page, using the `<a>` ("anchor") tag.

Another relevant element of HTML is tables. The `<table>` and `</table>` tags indicate the beginning and the end of the table, and its optional `border` attribute displays borders with a given width. The `<tr>` and `</tr>` tags introduce and close each row, and the tags

Continued on next page

`<th>…</th>` and `<td>…</td>` indicate a table header cell and a table data cell, respectively. The number of columns depends on the items in each row. Different rows, in fact, can have different numbers of items.

HTML was recently refined by the W3C to be more consistent, flexible, and interoperable with advanced systems such as XML; the new version is named XHTML (Extensible HTML). HTML and XHTML are the subject of many books, and you can find dozens of tutorials on them just by browsing the Web. A good, complete source on the topic is *Mastering XHTML* by Tittel et al. (Sybex, 2001).

Delphi's HTML Producer Components

If your version of Delphi includes the HTML producer components (available on the Internet page of the Component Palette), you can use them to generate the HTML files and particularly to turn a database table into an HTML table. Many developers believe that the use of these components makes sense only when writing a Web server extension. Although they were introduced for this purpose and are part of the WebBroker technology, you can still use three out of the four producer components in any application in which you must generate a static HTML file.

Before looking at the HtmlProd example, which demonstrates the use of these HTML producer components, let me summarize their role:

- The simplest of the HTML producer components is the PageProducer, which manipulates an HTML file in which you've embedded special tags. The advantage of this approach is that you can generate such a file using the HTML editor you prefer. At run time, the PageProducer converts the special tags to actual HTML code, giving you a straightforward method for modifying sections of an HTML document. The special tags have the basic format `<#tagname>`, but you can also supply named parameters within the tag. You'll process the tags in the `OnTag` event handler of the PageProducer.
- The DataSetPageProducer extends the PageProducer by automatically replacing tags corresponding to field names of a connected data source.
- The DataSetTableProducer component is generally useful for displaying the contents of a table, query, or other dataset. The idea is to produce an HTML table from a dataset, in a simple yet flexible way. The component has a very nice preview, so you can see how the HTML output will look in a browser directly at design time.
- The QueryTableProducer is similar to the previous one (it is actually a subclass), but it's specifically tailored for building parametric queries based on input from an HTML

search form. For this reason, I'll delay the coverage of this component to the next chapter.

Producing HTML Pages

A very simple example of using tags is creating an HTML file that displays fields with the current date or a date computed relative to the current date, such as an expiration date. If you examine the HtmlProd example, you'll find the following component in the main form:

```
object PageProducer1: TPageProducer
  HTMLDoc.Strings = (...)
  OnHTMLTag = PageProducer1HTMLTag
end
```

The source HTML can be specified using an external file (with the advantage that you can edit it without having to recompile the application using it) or a string list, stored in the `HTMLDoc` property. This is a plain HTML file that might contain a few special tags introduced by the # symbol:

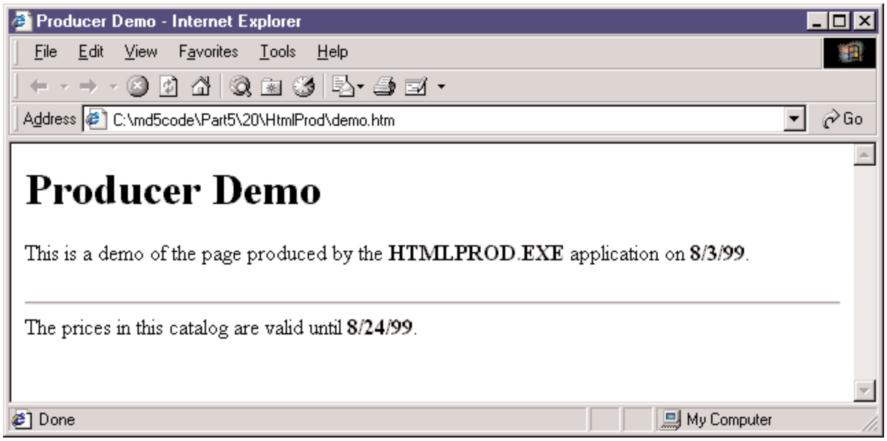
```
<html>
<head>
<title>Producer Demo</title>
</head>
<body>
<h1>Producer Demo</h1>
<p>This is a demo of the page produced by the <b><#appname></b> application on
<b><#date></b>.</p>
<hr>
<p>The prices in this catalog are valid until <b><#expiration
days=21></b>.</p>
</body>
</html>
```

WARNING If you prepare this file with an HTML editor (something I suggest you do), it might automatically place quotes around tag parameters, as in `days="21"`, because this is required by HTML 4 and XHTML 1. The PageProducer component has a `StripParamQuotes` property, which can be activated to remove those extra quotes when the component parses the code (before calling the `OnHTMLTag` event handler).

The Demo Page button simply copies the PageProducer component's output to the `Text` of a Memo with the statement

```
Memo1.Text := PageProducer1.Content;
```

As you call the `Content` function of the PageProducer component, it reads the input HTML code, parses it, and triggers the `OnTag` event handler for every special tag. In this method, we check the value of the tag (passed in the `TagString` parameter) and return a different HTML text (in the `ReplaceText` reference parameter), producing the output of Figure 21.8.


Producer Demo - Internet Explorer
File Edit View Favorites Tools Help
Address C:\md5code\Part5\20\HtmlProd\demo.htm
Go
Done
My Computer

Producer Demo

This is a demo of the page produced by the **HTMLPROD.EXE** application on **8/3/99**.

The prices in this catalog are valid until **8/24/99**.

FIGURE 21.8: The output of the HtmlProd example, a simple demonstration of the PageProducer component, when the user clicks the Demo Page button

```
procedure TFormProd.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
var
  nDays: Integer;
begin
  if TagString = 'date' then
    ReplaceText := DateToStr (Now)
  else if TagString = 'appname' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'expiration' then
  begin
    nDays := StrToIntDef (TagParams.Values['days'], 0);
    if nDays <> 0 then
      ReplaceText := DateToStr (Now + nDays)
    else
      ReplaceText := '<i>{expiration tag error}</i>';
  end;
end;
```

Notice, in particular, the code we've written to convert the last tag, `#expiration`, which requires a parameter. The PageProducer places the entire text of the tag parameter (in this case, *days=21*) in a string that's part of the `TagParams` list. To extract the value portion of this string (the portion after the equal sign), you can use the `Values` property of the `TagParams` string list and search for the proper entry at the same time. If it can't locate the parameter or if its value isn't an integer, the DLL displays an error message.

TIP The PageProducer component supports user-defined tags, which can be any string you like, but you should first review the special tags defined by the `TTags` enumeration. The possible values include tgLink (for the `link` tag), tgImage (for the `img` tag), tgTable (for the `table` tag), and a few others. If you create a custom tag, as in the PageProd example, the value of the `Tag` parameter to the HTMLTag handler will be tgCustom.

Producing Pages of Data

The HtmlProd example also has a DataSetPageProducer component, with the following settings and HTML source code:

```
object DataSetPageProducer1: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<html><head>'
    '<title>Data for <#name></title>'
    '</head><body>'
    '<h1><center>Data for <#name></center></h1>'
    '<p>Capital: <#capital></p>'
    '<p>Continent: <#continent></p>'
    '<p>Area: <#area></p>'
    '<p>Population: <#population></p>'
    '<hr>'
    '<p>Last updated on <#date><br>'
    'HTML file produced by the program <#program>.</p>'
    '</body></html>')
  OnHTMLTag = DataSetPageProducer1HTMLTag
  DataSet = Table1
end
```

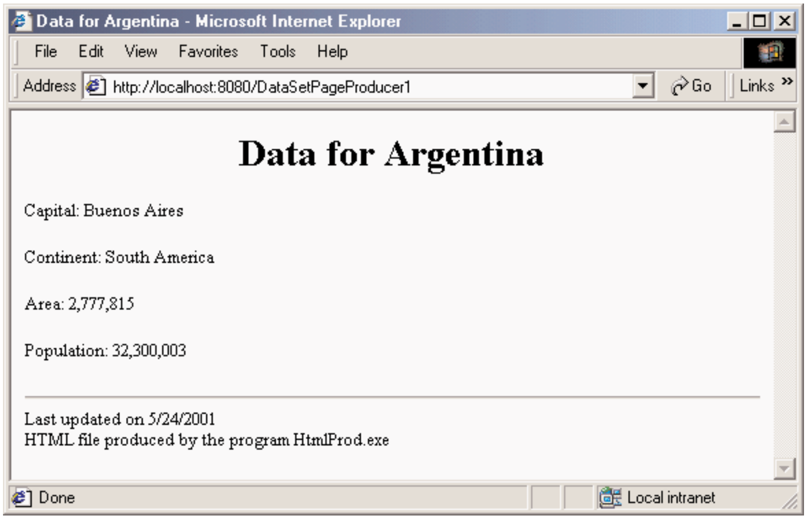
Simply by using tags with the names of the fields of the connected dataset (the usual COUNTRY.DB database table), the program automatically gets the value of the fields of the current record and replaces it automatically. This produces the output of Figure 21.9, which shows a browser connected to the HtmlProd example working as an HTTP server, as I'll discuss later. In the source code of the program related to this component, in fact, there is no reference to the database data:

```
procedure TFormProd.BtnLineClick(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.Text := DataSetPageProducer1.Content;
  BtnSave.Enabled := True;
end;

procedure TFormProd.DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
```

```
begin
  if TagString = 'program' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'date' then
    ReplaceText := DateToStr (Date);
end;
```

FIGURE 21.9:

The output of the HtmlProd example for the Print Line button


Data for Argentina - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:8080/DataSetPageProducer1
Go
Links
Done
Local intranet

Data for Argentina

Capital: Buenos Aires

Continent: South America

Area: 2,777,815

Population: 32,300,003

Last updated on 5/24/2001
HTML file produced by the program HtmlProd.exe

Producing HTML Tables

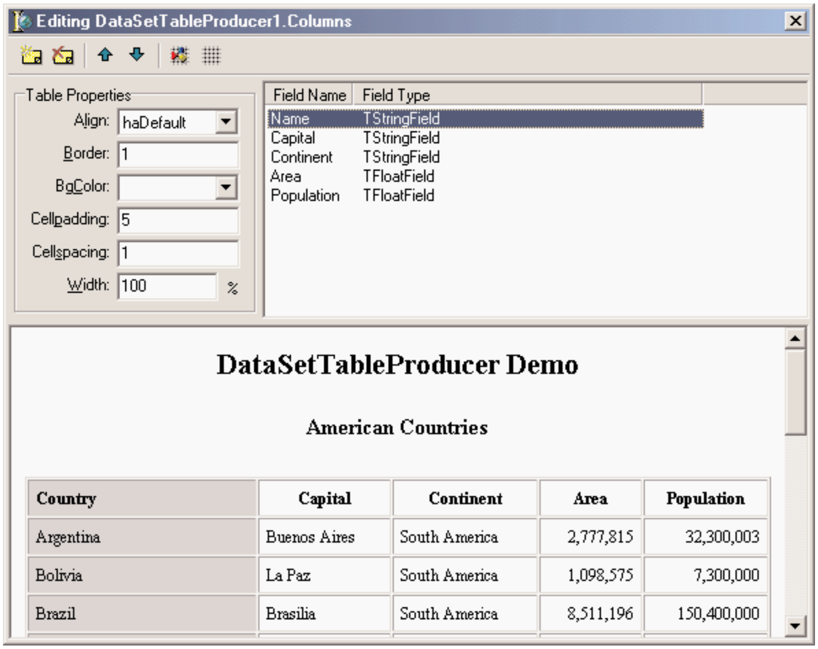
The last button of the HtmlProd example is Print Table. This button is connected to a DataSetTableProducer component, again calling its `Content` function and copying its result to the `Text` of the Memo. By simply connecting the `DataSet` property of the DataSetTableProducer to `Table1`, you can produce a standard HTML table. Actually, the component by default generates only 20 rows, as indicated by the `MaxRows` property. If you want to get all the records of the table, you can set this property to `-1`, a simple but undocumented setting.

TIP The DataSetTableProducer component starts from the current record rather than from the first one. This means that the second time you press the Print Table button, you'll see no records in the output. Adding a call to the `First` method of the table before calling the `Content` method of the producer component fixes the problem.

To make the output of this producer component more complete, you can do two different operations. The first is to provide some `Header` and `Footer` information, to generate the HTML heading and closing elements, and add a `Caption` to the HTML table. The second is to customize the table itself, by using the setting specified by the `RowAttributes`, `TableAttributes`, and `Columns` properties. The property editor of the columns, which is also the default component editor, allows you to set most of these properties, providing at the same time a very nice preview of the output, as you can see in Figure 21.10. Before using this editor, you can set up properties for fields of the table, using the Fields editor. This is how, for example, you can format the output of the population and area fields to use thousands separators.

FIGURE 21.10:

The editor of the `Columns` property of the DataSetTableProducer component provides you with a preview of the final HTML table (if the database table is active).



Editing DataSetTableProducer1.Columns

Table Properties

Align: haDefault

Border: 1

BgColor:

Cellpadding: 5

Cellspacing: 1

Width: 100 %

Field Name	Field Type
Name	TStringField
Capital	TStringField
Continent	TStringField
Area	TFloatField
Population	TFloatField

DataSetTableProducer Demo

American Countries

Country	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2,777,815	32,300,003
Bolivia	La Paz	South America	1,098,575	7,300,000
Brazil	Brasilia	South America	8,511,196	150,400,000

There are three techniques you can use to customize the HTML table, and it's worth reviewing each of them:

- You can use the table producer component's `Column` property to set properties, such as the text and color of the title, or the color and the alignment for the cells in the rest of the column.
- You can use the `TField` properties, particularly those related to output. In the example, I've set the `DisplayFormat` property of the `Table1Continent` field object to `###,###,###`. This is the approach to use if you want to determine the actual output of each field. You might go even further and embed HTML tags in the output of a field.
- You can handle the DataSetTableProducer component's `OnFormatCell` event to customize the output further. In this event, you can set the various column attributes

uniquely for a given cell, but you can also customize the output string (stored in the CellData parameter) and embed HTML tags. This is something you can't do using the Columns property.

In the HtmlProd example, I've used a handler for this event to turn the text of the Population and Area columns to bold font and to a red background for large values (unless it is the header row). Here is the code:

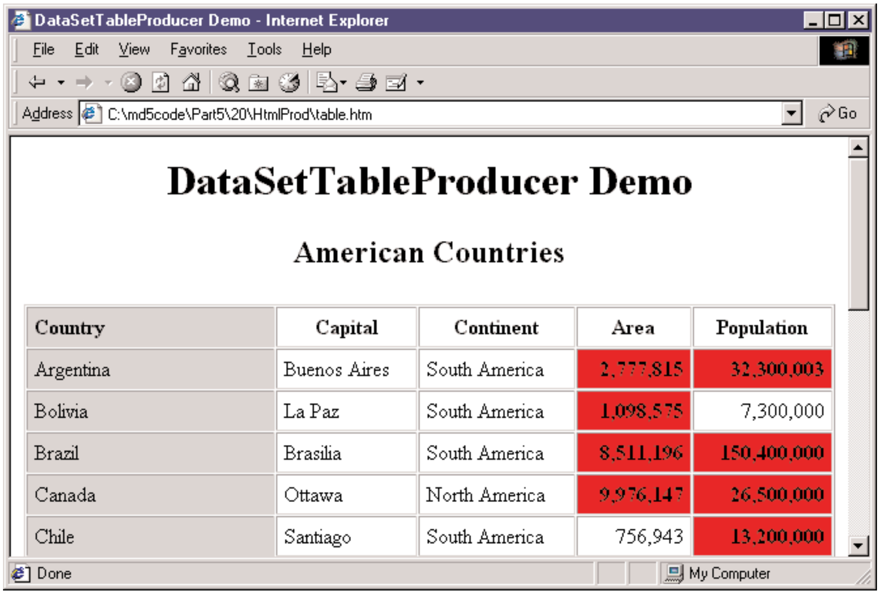
```
procedure TFormProd.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and
    (((CellColumn = 3) and (Length (CellData) > 8)) or
    ((CellColumn = 4) and (Length (CellData) > 9))) then
  begin
    BgColor := 'red';
    CellData := '<b>' + CellData + '</b>';
  end;
end;
```

The rest of the code is summarized by the settings of the table producer component (formatted slightly to make it more readable and take less space):

```
object DataSetTableProducer1: TDataSetTableProducer
  Caption = '<h2>American Countries</h2>'
  Columns = <
    item FieldName = 'Name'
      BgColor = 'silver'
      Title.Align = haLeft
      Title.BgColor = 'silver'
      Title.Caption = 'Country'
    end
    item FieldName = 'Capital'...
    item FieldName = 'Continent'...
    item FieldName = 'Area'
      Align = haRight
    end
    item FieldName = 'Population'
      Align = haRight
    end>
  Footer.Strings = ('<hr><i>Produced by HtmlProd</i></body></html>')
  Header.Strings = (<html><head><title>DataSetTableProducer Demo</title>'
    '</head><body><h1><center>DataSetTableProducer Demo</center></h1>')
  MaxRows = -1
  DataSet = Table1
```

```
    TableAttributes.Border = 1
    TableAttributes.CellPadding = 5
    OnFormatCell = DataSetTableProducer1FormatCell
  end
```

You can see the output of this program in Figure 21.11. I suggest you study the source code of the HTML file this program generates so that you can see the richness of its output and therefore the advantage of using this component.



C:\md5code\Part5\20\HtmlProd\table.htm

DataSetTableProducer Demo

American Countries

Country	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2,777,815	32,300,003
Bolivia	La Paz	South America	1,098,575	7,300,000
Brazil	Brasilia	South America	8,511,196	150,400,000
Canada	Ottawa	North America	9,976,147	26,500,000
Chile	Santiago	South America	756,943	13,200,000

FIGURE 21.11: The output of the Print All button of the HtmlProd example, which is based on the DataSetTableProducer component

Using Style Sheets

The latest incarnations of HTML include a very powerful mechanism for separating content from presentation: Cascading Style Sheets (CSS). Using a style sheet, you can separate the formatting of the HTML (colors, fonts, font sizes, and so on) from the actual text displayed (the content of the page). This approach makes your code more flexible and your Web site easier to update. In addition, you can separate the task of making the site graphically appealing (the work of a Web designer) from automatic content generation (the work of a programmer). Style sheets are a rather complex technique, in which you give formatting values to the main types of HTML sections and to special “classes” (which have nothing to do with OOP). Again, see an HTML reference for the details.

How can we update table generation in the HtmlProd example to include style sheets? Simply enough, we can provide a link to the style sheet to use in the Header property of a second DataSetTableProducer component, with the line

```
<link rel="stylesheet" type="text/css" href="test.css">
```

We can then update the code of the OnFormatCell event handler with the following action (instead of the two lines changing the color and adding the bold font tag):

```
CustomAttrs := 'class="highlight"';
```

The style sheet I've provided (test.css, available in the source code of the example) defines a *highlight* style, which has exactly the bold font and red background that were hard-coded in the first DataSetTableProducer component.

The advantage of this approach is that now a graphic artist can modify the CSS file and give our table a nicer look without touching its code. When you want to provide many formatting elements, using a style sheet can also reduce the total size of the HTML file. This is an important element that can reduce download time.

Dynamic Pages from a Custom Server

The HtmlProd component can be used to generate static HTML files, but doubles as a Web server, using an approach similar to what I've demonstrated in the HttpServ example, but in a more realistic context. The program, in fact, accesses the request of one of the possible page producers, simply passing the name of the component in a request. This is a portion of the OnCommandGet event handler of its IdHTTPServer component, which uses the FindComponent method to locate the proper producer component:

```
var
  Req, Html: String;
  Comp: TComponent;
begin
  Req := RequestInfo.Document;
  if Req [1] = '/' then
    Req := Copy (Req, 2, 1000); // skip '/'
  Comp := FindComponent (Req);
  if (Req <> '') and Assigned (Comp) and
    (Comp is TCustomContentProducer) then
  begin
    Table1.First;
    Html := TCustomContentProducer (Comp).Content;
  end;
  ResponseInfo.ContentText := Html;
end;
```

In case the parameter is not there (or is not valid), the server responds with an HTML-based menu of the available components:

```
Html := '<h1>Html Proc Menu<h1><p><ul>';
for I := 0 to ComponentCount - 1 do
  if Components [i] is TCustomContentProducer then
    Html := Html + '<li><a href="/' + Components [i].Name + '">' +
      Components [i].Name + '</a></li>';
Html := Html + '</ul></p>';
```

Finally, if the program returns a table that uses CSS, the browser will request the CSS file from the server, so I've added some specific code to return it. With the proper generalizations, this code shows how a server can respond, returning files, and also how to indicate the MIME type of the response (ContentType):

```
if Pos ('test.css', Req) > 0 then
begin
  CssTest := TStringList.Create;
  try
    CssTest.LoadFromFile(ExtractFilePath(Application.ExeName) + 'test.css');
    ResponseInfo.ContentText := CssTest.Text;
    ResponseInfo.ContentType := 'text/css';
  finally
    CssTest.Free;
  end;
  Exit;
end;
```

Publishing Static Databases on the Web

Once you know how to produce files, you can simply add links from one to another and produce a series of cross-linked HTML files, representing a portion of a Web site. There are circumstances in which writing a program that examines a database and produces files is the best approach for publishing database data on a Web site. You can use a similar technique if the following conditions apply:

If the data doesn't change very often A catalogue updated monthly or weekly is a good example. Even if you can update the site automatically every night, this is still a possible technique. (For real-time information, of course, this is certainly not a good approach!)

If the amount of data is limited and available space not is an issue This seems obvious, but the formatted HTML output might take much more space than the original database files. If you use a server-side program (such as those I'll be discussing in the next chapter) to generate the HTML from the database data on the fly, you might need less disk space on the Web site. Keep in mind that preparing all the HTML files beforehand usually results in much better performance (faster server response time to Web requests, and lower memory overhead to process the requests) than generating the data on the fly.

If the number of ways to navigate is limited If there are three or four obvious paths of navigation (a main one and two or three cross-references), you can generate all of them statically. Otherwise, the cross-referencing HTML files will be much larger than the files with the actual data, and the time required to generate them may become excessive.

Even if only parts of these conditions apply to your specific needs, you can consider using a mixed approach. You can have a portion of the data and of the navigational files generated periodically and have a CGI and ISAPI application on the site, as well as let users do free searches and follow other less frequent paths.

NOTE On the companion CD you can find an example, called DbCross, that generates hundreds of HTML files out of a master/detail database structure. The program also collects other data about the records, while generating the files, and produces a complete cross-reference. At the end, you can navigate by customers–orders by each customer–details of the order or by sale parts–orders where each part appears–details of each order. The code is quite complex, but I've tried to comment it in some detail, so you should be able to follow it.

What's Next?

In this chapter, we've focused on some core Internet technologies, including the use of sockets and core Internet protocols. I've discussed the main idea and shown a few examples of the use of the mail and HTTP protocols. You can find many more examples of the use of the Indy components in the demos done by their developers.

After this introduction to the world of the Internet, we are now ready to delve into two key areas, the present and the future. The present is represented by the development of Web applications, and we'll explore the development of dynamic Web sites in the next chapter, focusing first on the *old* WebBroker technology and then moving to the new WebSnap architecture. The future is represented by the development of Web services and the use of XML and related technology, which will be discussed in Chapter 23.

CHAPTER 22

Web Programming with WebBroker and WebSnap

- Dynamic Web pages
- CGI, ISAPI, and Apache modules
- The WebBroker architecture
- The Web App debugger
- The new WebSnap architecture
- Adapters and server-side scripting

If the Internet has a growing role in the world, a good part of it depends on the success of the World Wide Web, based on the HTTP protocol. We've already discussed, in the preceding chapter, HTTP and the development of client- and server-side applications based on it. With the availability of several high-performance, scalable, and flexible Web servers, you'll rarely want to create your own. Dynamic Web server applications, in fact, are generally built by integrating scripting or compiled programs within Web servers, rather then replacing them with custom software.

This chapter is entirely focused on the development of server-side applications, which extend existing Web servers. We have already introduced the dynamic generation of HTML pages toward the end of the last chapter. Now we have to see how to integrate this dynamic generation within a server. This chapter is a logical continuation of the last one but won't complete the coverage of Internet programming, as the next chapter is further devoted to this topic, covering specifically XML and Web services.

WARNING To test some of the examples in this chapter, you'll need access to a Web server. The simplest solution is probably to use the version of Microsoft's IIS or Personal Web Server already installed on your computer. My personal preference, however, is to use the free and open-source Apache Web Server, available (along with extensive documentation) at `www.apache.org`. In any case, I won't spend much time giving you details on the configuration of your Web server to enable the use of applications; refer to its documentation for this.

Dynamic Web Pages

When you browse a Web site, you generally download static pages—HTML-format text files—from the Web server to your client computer. As a Web developer, you can create these pages manually, but for most businesses, it makes more sense to build the static pages from information in a database of some type (a SQL server, a series of files, and so on). Using this approach, you're basically generating a snapshot of the data in HTML format, which is quite reasonable if the data isn't subject to frequent changes. This approach was discussed in Chapter 21, "Internet Programming: Sockets and Indy Components."

As an alternative to static HTML pages, you can build dynamic ones. To do this, you extract information directly from a database in response to the browser's request, so that the HTML sent by your application displays current data, not an old snapshot of the data. This approach makes sense if the data changes frequently.

As mentioned earlier, there are a couple of ways you can program custom behavior at the Web server, and these are ideal ways for you to generate HTML pages dynamically. The two most common protocols for programming Web servers are CGI (the Common Gateway Interface) and the Web server APIs. Another technique, Active Server Pages (ASP), is quite popular in the Microsoft world, and I'll discuss it briefly because Delphi includes specific support for it.

NOTE Keep in mind that Delphi's WebBroker technology (available in both the Enterprise and Professional editions) flattens the differences between CGI, WinCGI, and ISAPI by providing a common class framework. This way, you can easily turn a CGI application into a WinCGI one, upgrade it to use the ISAPI model, or integrate it into Apache.

An Overview of CGI

CGI is a standard protocol for communication between the client browser and the Web server. It's not a particularly efficient protocol, but it is widely used and is not platform specific. This protocol allows the browser both to ask for and to send data, and it is based on the standard command-line input and output of an application (usually a console application). When the server detects a page request for the CGI application, it launches the application, passes command-line data from the page request to the application, and then sends the standard output of the application back to the client computer.

There are many tools and languages you can use to write CGI applications, and Delphi is only one of them. Given the obvious limitation that your Web server must be an Intel-based Windows or Linux system, you can build some fairly sophisticated CGI programs in Delphi and Kylix. Despite the fact that it's called a standard, there are actually different flavors of CGI. Traditional CGI uses the standard command-line input and output, along with environment variables. WinCGI uses an INI file passed as a command-line parameter to the application (instead of environment variables) and specific input and output files (instead of using command-line input/output). Server vendors developed WinCGI primarily for Visual Basic programmers, who cannot access environment variables. Another new variation, called FastCGI, is supposed to make the entire process of calling a CGI application much faster, but it's not widely supported yet.

To build a CGI program without using any support class, you can simply create a Delphi console application, remove the typical project source code, and replace it with the following statements:

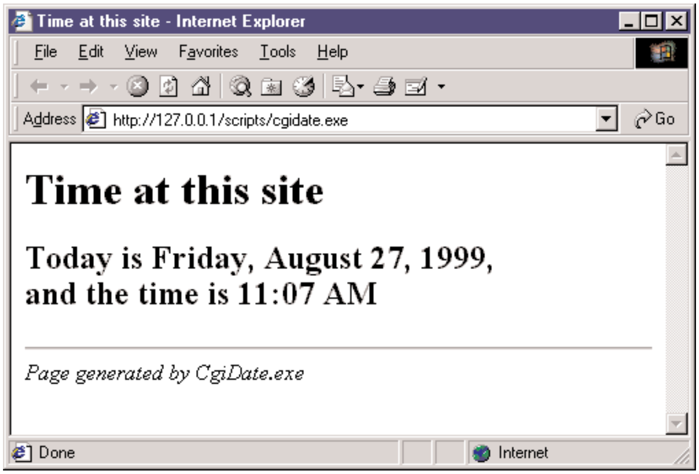
```
program CgiDate;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
```

```
  writeln ('content-type: text/html');
  writeln;
  writeln ('<html><head>');
  writeln ('<title>Time at this site</title>');
  writeln ('</head><body>');
  writeln ('<h1>Time at this site</h1>');
  writeln ('<hr>');
  writeln ('<h3>');
  writeln (FormatDateTime('"Today is " dddd, mmmm d, yyyy,' +
    '"<br> and the time is" hh:mm:ss AM/PM', Now));
  writeln ('</h3>');
  writeln ('<hr>');
  writeln ('<i>Page generated by CgiDate.exe</i>');
  writeln ('</body></html>');
end.
```

CGI programs produce a header followed by the HTML text using the standard output. If you execute this program directly, you'll see the text in a terminal window. If you run it instead from a Web server and send the output to a browser, the formatted HTML text will appear, as shown in Figure 22.1.

FIGURE 22.1:

The output of the CgiDate application, as seen in a browser


Time at this site - Internet Explorer
File
Edit
View
Favorites
Tools
Help
Address
http://127.0.0.1/scripts/cgidate.exe
Go
Done
Internet

Time at this site

Today is Friday, August 27, 1999,
and the time is 11:07 AM

Page generated by CgiDate.exe

Building advanced and complex applications with plain CGI requires a lot of work. For example, to extract status information on the HTTP request, you need to access to the relevant environment variables, as in:

```
// get the pathname
GetEnvironmentVariable ('PATH_INFO', PathName, sizeof (PathName));
```

An Overview of ISAPI/NSAPI

A completely different approach is the use of the Web server APIs, the popular ISAPI (Internet Server API, introduced by Microsoft) and the less common NSAPI (Netscape Server API). These APIs allow you to write a DLL that the server loads into its own address space and usually keeps in memory for some time. Once it loads the DLL, the server can execute individual requests via threads within the main process, instead of launching a new EXE for every request (as it must in CGI applications).

When the server receives a page request, it loads the DLL (if it hasn't done so already) and executes the appropriate code, which may launch a new thread or use an existing one to process the page request (the IIS Web server offers thread pooling support to avoid creating a new thread for each request). The DLL code then sends the appropriate data back to the client that requested the page. Because this communication generally occurs in memory, this type of application is much faster than CGI, and a given system will be able to support more simultaneous page requests this way.

The main drawback to server API DLLs is that their tight integration with the server is an Achilles' heel; if the DLL crashes or produces memory leaks, the entire Web server can crash. However, the most recent releases of Microsoft's IIS Web server fix the problem by running the DLL in a *protected* space. Another problem is that when the DLL is in memory, you cannot compile an updated version; you need to unload the DLL first or momentarily stop the Web server (an operation you can do only on a test-bed computer).

Technically, ISAPI DLLs are not very different from plain Windows DLLs. They must export a couple of specific functions that the Web server will call: `GetExtensionVersion` and `HttpExtensionProc`. The server calls the first function when it loads the DLL for the first time and the second function for every following request. The parameters of these functions are complex data structures holding input data and server methods you can call to produce the result. Here is a sample of this function (taken from the IsapiDem example), which uses the `lpszPathInfo` field and the `WriteClient` function:

```
function HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD; stdcall;
var
  OutStr: string;
  StrLength: Cardinal;
begin
  with ECB do
  begin
    OutStr :=
      '<html><head><title>First Isapi Demo</title></head><body>' +
      '<h2><center>First Isapi Demo</center></h2>' +
      '<p>Hello Mastering Delphi Readers...</p><hr>' +
      '<p><b>Activated by ' + PChar(@lpszPathInfo[1]) + '</b></p>' +
      '<p><i>From IsapiDLL on ' + DateToStr(Now) + ' at ' + TimeToStr(Now) +
      '</i></p></body></html>';
    StrLength := Length(OutStr);
    WriteClient(ConnID, PChar (OutStr), StrLength, 0);
  end;
  Result := HSE_STATUS_SUCCESS;
end;
```

The program doesn't simply use the `lpszPathInfo` parameter but uses the substring starting with the second character, to get rid of the initial slash. To be more precise, the expression `PChar(@lpszPathInfo[1])` takes the string starting at the memory address of the second character of the path (a zero-based characters array).

Apache Modules

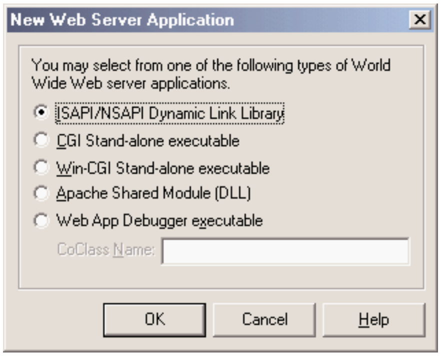
Similarly to Microsoft's IIS, the Apache server of the Apache Foundation (`www.apache.org`) allows server-side extensions by means of CGI or with specific extension libraries. In case of Apache, these libraries are called *modules*, or dynamic modules. In the Apache configuration, you can list the modules you are interested in and eventually connect them to a virtual directory.

Needless to say, you can program Apache modules in a similar low-level way to what I've just done for ISAPI, but I won't show you an example. I'll do it later using the Apache support added to the WebBroker architecture in Delphi 6.

Delphi's WebBroker Technology

The CGI and ISAPI code snippets I've shown you so far demonstrate the plain, direct approach to the protocol and API. Extending these examples at that level is certainly possible, but what is interesting in Delphi is to use the WebBroker technology. This comprises a class hierarchy within VCL and CLX, built to simplify server-side development on the Web, and a specific type of data modules, called WebModules. Both the Enterprise and Professional editions of Delphi 5 include this framework (differently from the more advanced and newer WebSnap framework, which is available only in the Enterprise version of Delphi 6).

Using the WebBroker technology, you can begin developing an ISAPI or CGI application or an Apache module very easily. On the first page (*New*) of the New Items dialog box, select the Web Server Application icon. The subsequent dialog box will offer you options (two more than in Delphi 5). As you can see in Figure 22.2, you can choose ISAPI, CGI, WinCGI, Apache module, and the Web App Debugger.



New Web Server Application

You may select from one of the following types of World Wide Web server applications.

● ISAPI/NSAPI Dynamic Link Library
○ CGI Stand-alone executable
○ Win-CGI Stand-alone executable
○ Apache Shared Module (DLL)
○ Web App Debugger executable

CoClass Name:

FIGURE 22.2:

The alternative options for building a Web server application in Delphi

TIP As a starting point for your server-side application, you can also use the DB Web Application Wizard, available in the Business page of the New Items dialog box. This wizard generates a program with a BDE table or query connected to a DataSetTableProducer. It can be helpful, but the generated code is really very limited and there is no support for Apache and the Web App Debugger.

For example, if you select the first option, Delphi will generate the basic structure of an ISAPI application for you. The application that Delphi generates (no matter which type you choose) is based on the `TWebModule` class, a container very similar to a data module. The WebModule code is similar to that of a data module, as we'll see in a moment, but the code of the library is worth looking at:

```
library Project1;

uses
  WebBroker,
  ISAPIThreadPool,
  ISAPIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

{$R *.RES}

exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

WARNING This code changes slightly between Delphi versions. If you have older code around, you'll need to refer to the WebBroker unit instead of the previous HTTPApp unit. Delphi 6 adds the reference to the ISAPIThreadPool unit, which provides support for pooling threads under ISAPI (with a couple of classes not documented in the Help file).

Although this is a library that exports the ISAPI functions, the code looks similar to that of an application. However, it uses a trick—the `Application` object used by this program is not the typical global object of class `TApplication` but an object of a new class. This new `Application` object is of class `TISAPIApplication` (or `TCGIApplication` if you've built that type of application), which derives from `TWebApplication`.

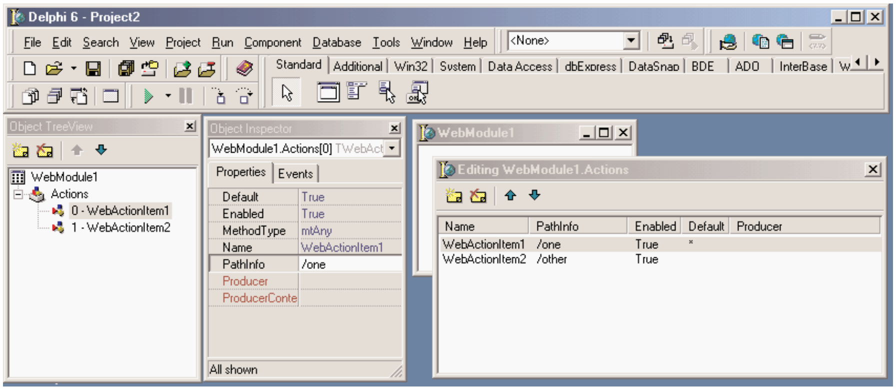
Although these application classes provide the foundations, you won't use them very often (just as you don't use the `Application` object very often in a form-based Delphi application). The most important operations take place in the WebModule. This component derives from `TCustomWebDispatcher`, which provides support for all the input and output of our programs.

In fact, the `TCustomWebDispatcher` class defines `Request` and `Response` properties, which store the client request and the response we're going to send back to the client. Each of these properties is defined using a base abstract class (`TWebRequest` and `TWebResponse`), but an application initializes them using a specific object (such as the `TISAPIRequest` and `TISAPIResponse` subclasses). These classes make available all the information passed to the server, so you have a single, simple approach to accessing all the information. The same is true of a response, which is very easy to manipulate. One advantage to this approach is that an ISAPI DLL written with this framework is very similar to a CGI application; in fact, they are frequently identical in the source code you write.

If this is the structure of Delphi's framework, how do you write the application code? Well, in the WebModule, you can use the Actions editor (shown in Figure 22.3) to define a series of actions (stored in the `Actions` array property) depending on the *pathname* of the request. This pathname is a portion of the CGI or ISAPI application's URL, which comes after the program name and before the parameters, such as `path1` in the following URL:

```
http://www.example.com/scripts/cgitest.exe/path1?param1=date
```

By providing different actions, your application can easily respond to requests with different pathnames, and you can assign a different producer component or call a different `OnAction` event handler for every possible pathname. Of course, you can omit the pathname to handle a generic request. Consider also that, instead of basing your application on a WebModule, you can use a plain data module and add a WebDispatcher component to it. This is a good approach if you want to turn an existing Delphi application into a Web server extension.


Delphi 6 - Project2
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access dbExpress DataSnap BDE ADO InterBase W
Object TreeView
Object Inspector
WebModule1
Editing WebModule1.Actions

WebModule1
- Actions
  - 0 - WebActionItem1
  - 1 - WebActionItem2

WebModule1.Actions[0] TWebAct

Properties | Events

Default	True
Enabled	True
MethodType	mtAny
Name	WebActionItem1
PathInfo	/one
Producer	
ProducerConte	

All shown

Name	PathInfo	Enabled	Default	Producer
WebActionItem1	/one	True	*	
WebActionItem2	/other	True		

FIGURE 22.3: The Actions property editor of the WebModule, along with the properties of one of the actions in the Object Inspector

WARNING The WebModule incorporates the WebDispatcher and doesn't require it as a separate component. Unlike WebSnap applications, in fact, WebBroker programs cannot have multiple dispatchers or multiple Web modules. Notice also that the actions of the WebDispatcher have absolutely nothing to do with the actions stored in a `TActionList` component.

When you define the accompanying HTML pages that launch the application, the links will make page requests to the URLs for each of those paths. Having one single ISAPI DLL that can perform different operations depending on a parameter (in this case, the pathname) allows the server to keep a copy of this DLL in memory and respond much faster to user requests. The same is partially true for a CGI application: The server has to run several instances but can cache the file and make it available faster.

The `OnAction` event is where you put the code to specify the *response* to a given *request*, the two main parameters passed to the event handler. Here is a simple example:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    '<html><head><title>Hello Page</title></head><body>' +
    '<h1>Hello</h1>' +
    '<hr><p><i>Page generated by Marco</i></p></body></html>';
end;
```

The `Content` property of the `Response` parameter is where you enter the HTML code that you want users to see. The only drawback of this code is that the output in a browser will be correctly displayed on multiple lines, but looking at the HTML source code, you'll see a single line corresponding with the entire string. To make the HTML source code more readable, by splitting it up onto multiple lines, you can insert the `#13` newline character.

To let other actions handle this request, you'll set the last parameter, `Handled`, to False. Otherwise, the default value is True, and once you've handled the request with your action, the WebModule assumes you're finished. Most of an ISAPI application's code will be in the `OnAction` event handlers for the actions defined in the WebModule container. These actions receive a request from the client and return a response using the `Request` and `Response` parameters.

When using the producer components, your `OnAction` event often returns, as `Response.Content`, the `Content` of the producer component, with a simple assignment. You can shortcut this code by assigning a producer component to the `Producer` property of the action itself, with no need to write these simple event handlers anymore (but don't do both things, as that might get you into trouble).

TIP As an alternative to the `Producer` property, you can use the `ProducerContent` property introduced in Delphi 6. This new property allows you to connect custom producer classes that don't inherit from the `TCustomContentProducer` class but implement the `IProduceContent` interface. The `ProducerContent` property is *almost* an interface property: It behaves in the same way, but thanks to its property editor and not based on the new support for interfaced properties of Delphi 6.

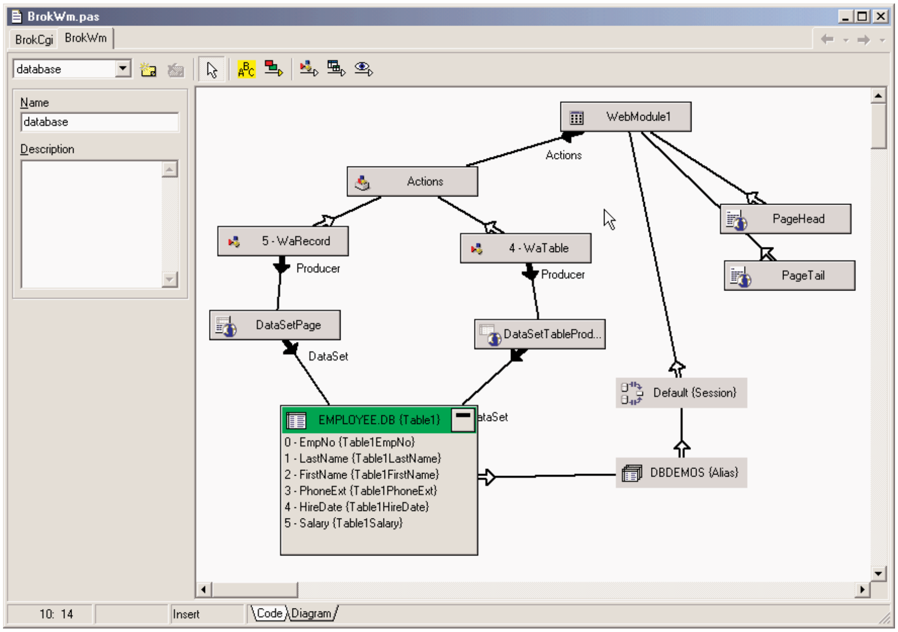
Building a Multipurpose WebModule

To demonstrate how easily you can build a feature-rich server-side application using Delphi's support, I've created the BrokDemo example. This example can be compiled as a CGI or an ISAPI application, simply by choosing the proper project file. The WebModule is shared by the two projects, without any difference in the source code, a practical proof that, using the WebBroker framework, you can move from ISAPI to CGI, from Apache to the Web App Debugger (although this last step requires a little extra tweaking in the source code). In the past, I tended to test programs with CGI (to avoid having to stop the server to free the library and recompile it) and then deploy them with ISAPI. Now I tend to test with the Web App Debugger and then deploy under Apache.

A key element is the list of actions we're going to support with this application. The actions can be managed in the Actions editor or directly in the Object TreeView, as we've already seen in Figure 22.3. Actions are also visible in the Designer page of the editor, so you can graphically see their relationship with database objects, as shown for the BrokDemo in Figure 22.4. If you examine the figure or the source code, you'll notice that I've given a specific name to every action. I've also given meaningful names to the `OnAction` event handlers. For instance, `TimeAction` as a method name should be much more understandable than the `WebModule1WebActionItem1Action` name automatically generated by Delphi.

FIGURE 22.4:

The structure of the BrokDemo example, as shown by the Designer



Every action has a different pathname, with one of them marked as default and executed even if no pathname is specified. The first interesting idea in this program is the use of two PageProducer components, used for the initial and final portion of every page, `PageHead` and `PageTail`. Centralizing this code makes it easier to modify it, particularly if it is based on external HTML files. The HTML produced by these components is added at the beginning and the end of the resulting HTML in the `OnAfterDispatch` event handler of the Web module:

```
procedure TWebModule1.WebModule1AfterDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageHead.Content + Response.Content + PageTail.Content;
end;
```

I'm adding the initial and final HTML at the end of the page generation simply because this allows the components to produce the HTML as if they were making all of it. Starting with some HTML in the `OnBeforeDispatch` event means that you cannot directly assign the producer components to the actions, or the producer component will override the `Content` you've already provided in the response. The `PageTail` component includes a custom tag for the script name, replaced by the following code, which uses the current request object available within the Web module:

```
procedure TWebModule1.PageTailHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'script' then
    ReplaceText := Request.ScriptName;
end;
```

This code is activated to expand the <#script> tag of the PageTail component's HTMLDoc property. The code of the time and date actions is straightforward. The really interesting part begins with the Menu path, which is the default action. In its OnAction event handler, the application uses a for loop to build a list of the available actions (using their names without the first two letters, which are always Wa in my example), providing a link to each of them with an anchor (an <a> tag):

```
procedure TWebModule1.MenuAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<h3>Menu</h3><ul>'#13;
  for I := 0 to Actions.Count - 1 do
    Response.Content := Response.Content + '<li> <a href="' +
      Request.ScriptName + Action[I].PathInfo + '"> ' +
      Copy (Action[I].Name, 3, 1000) + '</a>'#13;
  Response.Content := Response.Content + '</ul>';
end;
```

Another action of the BrokDemo example provides users with a list of the system settings related to the request, something that is quite useful for debugging. It is also instructive to learn how much information, and exactly what information, the HTTP protocol transfers from a browser to a Web server and vice versa. To produce this list, the program looks for the value of each property of the TWebRequest class, as this initial snippet demonstrates:

```
procedure TWebModule1.StatusAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<h3>Status</h3>'#13 +
    'Method: ' + Request.Method + '<br>'#13 +
    'ProtocolVersion: ' + Request.ProtocolVersion + '<br>'#13 +
    'URL: ' + Request.URL + '<br>'#13 +
    'Query: ' + Request.Query + '<br>'#13 + ...
```

Dynamic Database Reporting

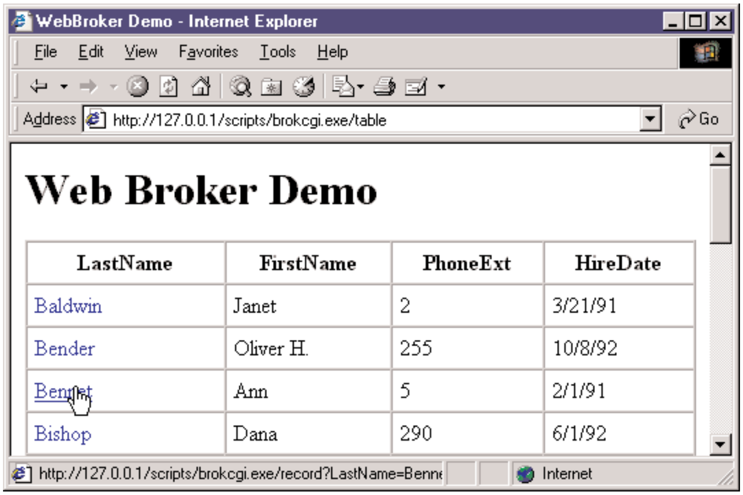
The BrokDemo example defines two more actions, indicated by the /table and /record pathnames. For these two last actions, our program produces a main list of names and then displays the details of one record, using a DataSetTableProducer component to format the entire table and a DataSetPageProducer component to build the record view. Here are the properties of these two components:

```
object DataSetTableProducer1: TDataSetTableProducer
  DataSet = Table1
  OnFormatCell = DataSetTableProducer1FormatCell
end
object DataSetPage: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<h3>Employee: <#LastName></h3>'
    '<ul><li> Employee ID: <#EmpNo>'
    '<li> Name: <#FirstName> <#LastName>'
    '<li> Phone: <#PhoneExt>'
    '<li> Hired On: <#HireDate>'
    '<li> Salary: <#Salary></ul>')
  OnHTMLTag = PageTailHTMLTag
  DataSet = Table1
end
```

To produce the entire table, we simply connect the DataSetTableProducer to the Producer property of the corresponding actions, without providing any specific event handler. The table is made more powerful by adding internal links to the specific records. The following code is executed for each cell of the table but activated only for the first column or the first row (the one with the title):

```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign;
  var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + ScriptName + '/record?LastName=' +
      Table1['LastName'] + '&FirstName=' + Table1 ['FirstName'] + '"> ' +
      CellData + ' </a>';
end;
```

You can see the result of this action in Figure 22.5. When the user selects one of the links, the program is called again, and it can check the QueryFields string list and extract the parameters from the URL. It then uses the values corresponding to the table fields used for the record search (which is based on the FindNearest call).


WebBroker Demo - Internet Explorer
File
Edit
View
Favorites
Tools
Help
Address
http://127.0.0.1/scripts/brokcgi.exe/table
Go
http://127.0.0.1/scripts/brokcgi.exe/record?LastName=Benne
Internet

Web Broker Demo

LastName	FirstName	PhoneExt	HireDate
Baldwin	Janet	2	3/21/91
Bender	Oliver H.	255	10/8/92
Bennet	Ann	5	2/1/91
Bishop	Dana	290	6/1/92

FIGURE 22.5: The output corresponding to the *table* path of the BrokDemo example, which produces an HTML table with internal hyperlinks.

```
procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Table1.Open;
  // go to the requested record
  Table1.FindNearest ([Request.QueryFields.Values['LastName'],
    Request.QueryFields.Values['FirstName']]);
  // get the output
  Response.Content := Response.Content + DataSetPage.Content;
end;
```

NOTE The example we've just built accesses a Paradox table via the BDE. The CGI version executes once for every request and will actually load and unload the BDE each time it runs. As alternatives, you can consider three different approaches: using ISAPI instead of CGI (to keep the application and the BDE loaded in memory), accessing the data from a plain file (or avoid the BDE with some other data access technology), or running another BDE application on the server (so that the BDE will remain loaded in memory). When using the BDE in an ISAPI application, though, you need to add a Session component to avoid concurrent access by multiple threads to the same BDE session.

Of Queries and Forms

The previous example used some of the HTML producer components introduced earlier in this chapter. There is another component of this group we haven't used yet, the QueryTableProducer. As we'll see in a moment, this component makes building even complex database programs a breeze. Suppose you want to search for some customers in a database. You might construct the following HTML form (embedded in an HTML table for better formatting):

```
<h4>Customer QueryProducer Search Form</h4>
<form action="/scripts/CustQueP.dll/search" method="POST">
<table>
<tr><td>State:</td>
  <td><input type="text" name="State"></td></tr>
<tr><td>Country:</td>
  <td><input type="text" name="Country"></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

NOTE

As in Delphi, an HTML form hosts a series of controls (typically, things like input fields). There are visual tools to help you design these forms, or you can manually enter the proper HTML code. The available controls include buttons, input text (or edit boxes), selections (or combo boxes), and radio buttons (or input buttons). You can define buttons as specific types, such as Submit or Reset, which imply standard behaviors. An important element of forms is the *request method,* which can be either POST (data is passed behind the scenes, and you receive it in the `ContentFields` property) or GET (data is passed as part of the URL, and you extract it from the `QueryFields` property).

There is a very important element to notice in the form: the names of the input components (*State* and *Country*), which should match the parameters of a Query component:

```
select
  Company, State, Country
from
  CUSTOMER.DB
where
  State = :State or Country = :Country
```

This code is used in the CustQueP (customer query producer) example. To build it, I've placed a Query component inside the WebModule and generated the field objects for it. In the same WebModule, I've added a QueryTableProducer component connected to the `Producer` property of the `/search` action. The program will generate the proper response. How does this work? When we activate the QueryTableProducer component by calling its `Content` function, it initializes the Query component by obtaining the parameters from the HTTP request. The component can automatically examine the request method and then use either the `QueryFields` property (if the request is a GET) or the `ContentFields` property (if the request is a POST).

One problem with using a static HTML form as we did before is that it doesn't tell us which states and countries we can search for. To address this, we can use a selection control instead of an edit control in the HTML form. However, if the user adds new records to the database table, we'll need to update the element list automatically. As a final solution, we can design the ISAPI DLL to produce a form on-the-fly, and we can fill the selection controls with the available elements.

We'll generate the HTML for this page in the /form action, which we've connected to a PageProducer component. The PageProducer contains the following HTML text, which embeds two special tags:

```
<h4>Customer QueryProducer Search Form</h4>
<form action="CustQueP.dll/search" method="POST">
<table>
<tr><td>State:</td>
  <td><select name="State"><#State></select></td></tr>
<tr><td>Country:</td>
  <td><select name="Country"><option> </option><#Country></select></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

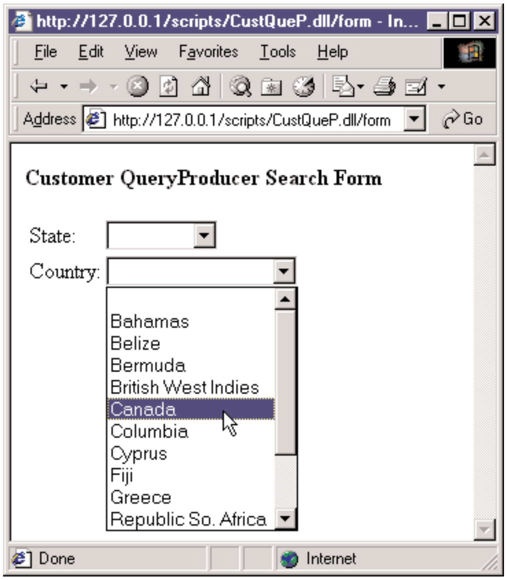
You'll notice that the tags have the same name as some of the table's fields. When the PageProducer encounters one of these tags, it adds an <option> HTML tag for every distinct value of the corresponding field. Here's the OnTag event handler's code, which is quite generic and reusable:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  ReplaceText := '';
  Query2.SQL.Clear;
  Query2.SQL.Add ('select distinct ' + TagString + ' from customer');
  try
    Query2.Open;
    try
      Query2.First;
      while not Query2.EOF do
      begin
        ReplaceText := ReplaceText +
          '<option>' + Query2.Fields[0].AsString + '</option>'#13;
        Query2.Next;
      end;
    finally
      Query2.Close;
    end;
  except
    ReplaceText := '{wrong field: ' + TagString + '}';
  end;
end;
```

This method used a second Query component, which I manually placed on the form and connected to the DBDEMOS database, and it produces the output shown in Figure 22.6.

FIGURE 22.6:

The form action of the CustQueP example produces an HTML form with a selection component dynamically updated to reflect the current status of the database.


http://127.0.0.1/scripts/CustQueP.dll/form - In...
File Edit View Favorites Tools Help
Address http://127.0.0.1/scripts/CustQueP.dll/form
Go
Done
Internet

Customer QueryProducer Search Form

State:

Country:

Bahamas
Belize
Bermuda
British West Indies
Canada
Columbia
Cyprus
Fiji
Greece
Republic So. Africa

Finally, this Web server extension, like many others we've built, allows the user to view the details of a specific record. As in the last example, we can accomplish this by customizing the output of the first column (column zero), which is generated by the QueryTableProducer component:

```
procedure TWebModule1.QueryTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + Request.ScriptName + '/record?' + CellData +
      '">' + CellData + '</a>'#13;
  if CellData = '' then
    CellData := '&nbsp;';
end;
```

TIP

When you have an empty cell in an HTML table, most browsers render it without the border. For this reason, I've added a "nonbreaking space" symbol (`&nbsp;`) into each empty cell. This is something you'll have to do in each HTML table generated with Delphi's table producers.

The action for this link is /record, and we'll pass a specific element after the ? parameter (without the parameter name, which is slightly nonstandard). The code we use to produce the HTML tables for the records doesn't use the producer components as we've been doing; instead, it is very similar to the code of an early ISAPI example:

```
procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  if Request.QueryFields.Count = 0 then
    Response.Content := 'Record not found'
  else
  begin
    Query2.SQL.Clear;
    Query2.SQL.Add ('select * from customer ' +
      'where Company="' + Request.QueryFields[0] + '"');
    Query2.Open;
    Response.Content :=
      '<html><head><title>Customer Record</title></head><body>'#13 +
      '<h1>Customer Record: ' + Request.QueryFields[0] + '</h1>'#13 +
      '<table border>'#13;
    for I := 1 to Query2.FieldCount - 1 do
      Response.Content := Response.Content +
        '<tr><td>' + Query2.Fields [I].FieldName + '</td>'#13'<td>' +
        Query2.Fields [I].AsString + '</td></tr>'#13;
    Response.Content := Response.Content + '</table><hr>'#13 +
      // pointer to the query form
      '<a href="' + Request.ScriptName + '/form">' +
      ' Next Query </a>'#13 + '</body></html>'#13;
  end;
end;
```

Debugging with the Web App Debugger

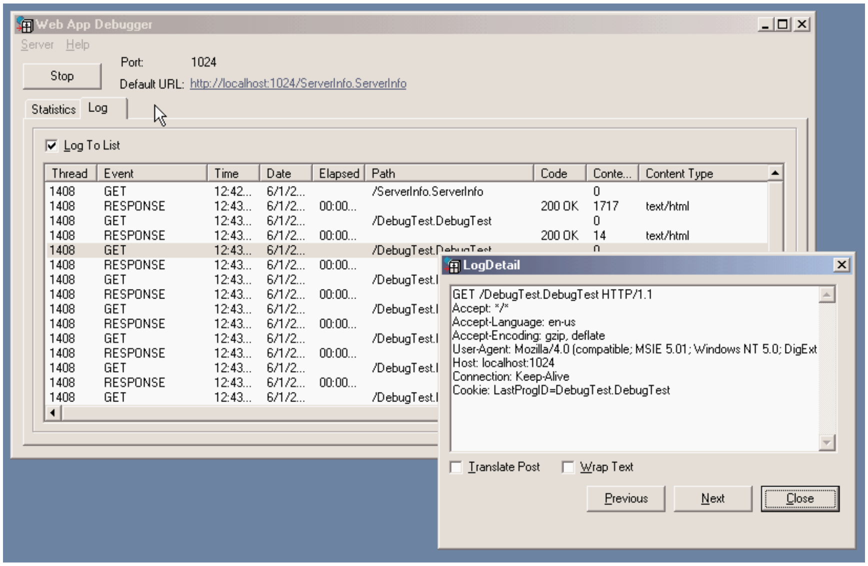
Debugging Web applications written in Delphi is often quite difficult. In fact, you cannot simply run the program and set breakpoints in it, but should convince the Web server to run your CGI program or library within the Delphi debugger. This can be accomplished by indicating a Host application in Delphi's Run Parameters dialog box, but it implies letting Delphi run the Web server (which is often a Windows service, not a stand-alone program).

To solve all of these issues, Borland has added to Delphi 6 a specific Web App Debugger program. This tool, activated by the corresponding item of the Tools menu, is a Web server, which waits for requests on a port you can set up (1024 by default). When a request arrives, the program can forward it to a stand-alone executable, using COM-based techniques. This means you can run the Web server application from within the Delphi IDE, set all the breakpoints you need, and then (when the program is activated through the Web App Debugger) debug the program as you'll do for a plain executable file.

The Web App Debugger does also a good job in logging all the received requests and the actual responses returned to the browser, as you can see in Figure 22.7. The program also has a Statistics page, which interestingly tracks the time required for each response, allowing you to test the efficiency of an application in different conditions.

FIGURE 22.7:

The log of the Web App Debugger with its LogDetail window



Port: 1024
Default URL: http://localhost:1024/ServerInfo.ServerInfo

☑ Log To List

Thread	Event	Time	Date	Elapsed	Path	Code	Conte...	Content Type
1408	GET	12:42...	6/1/2...		/ServerInfo.ServerInfo		0	
1408	RESPONSE	12:43...	6/1/2...	00:00...		200 OK	1717	text/html
1408	GET	12:43...	6/1/2...		/DebugTest.DebugTest		0	
1408	RESPONSE	12:43...	6/1/2...	00:00...		200 OK	14	text/html
1408	GET	12:43...	6/1/2...		/DebugTest.DebugTest		0	
1408	RESPONSE	12:43...	6/1/2...	00:00...				
1408	GET	12:43...	6/1/2...		/DebugTest.			
1408	RESPONSE	12:43...	6/1/2...	00:00...				
1408	GET	12:43...	6/1/2...		/DebugTest.			
1408	RESPONSE	12:43...	6/1/2...	00:00...				
1408	GET	12:43...	6/1/2...		/DebugTest.			
1408	RESPONSE	12:43...	6/1/2...	00:00...				
1408	GET	12:43...	6/1/2...		/DebugTest.			
1408	RESPONSE	12:43...	6/1/2...	00:00...				
1408	GET	12:43...	6/1/2...		/DebugTest.			

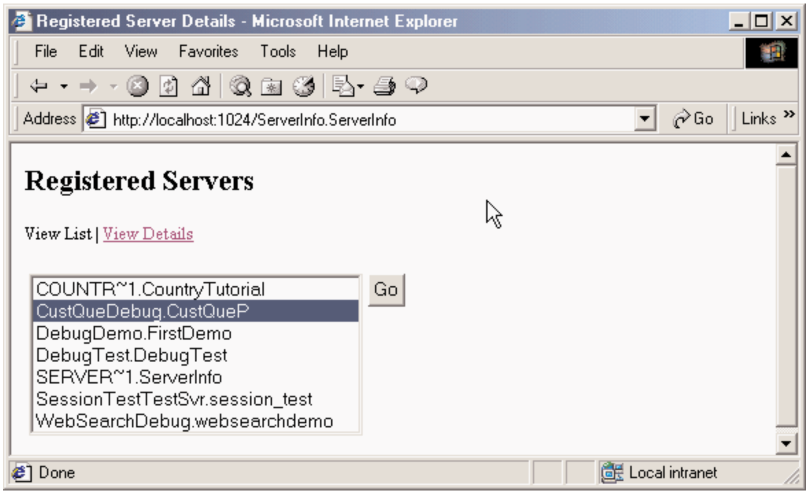
```
GET /DebugTest.DebugTest HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0; DigExt
Host: localhost:1024
Connection: Keep-Alive
Cookie: LastProgID=DebugTest.DebugTest
```

☐ Translate Post ☐ Wrap Text

By using the corresponding option of the New Web Server Application dialog, you can easily create a new application compatible with the debugger. This defines a standard project, which creates both a main form and a data module. The (useless) form includes code for registering the application as an OLE automation server, as:

```
const
  CLASS_ComWebApp: TGUID = '{33A4D4F0-E082-4723-9165-5D8F95AF1577}';

initialization
  TWebAppAutoObjectFactory.Create(Class_ComWebApp, 'FirstDemo',
    'FirstDemo Object');
```

The information is used by the Web App Debugger to get a list of the available programs. This is done when you use the default URL for the debugger, indicated in the form as a link, as you can see (for example) in Figure 22.8. The list includes all of the registered servers, not only those running. In fact, the use of COM Automation accounts for the automatic activation of a server. Not that this is a good idea, though, as running and terminating the program each time will make the process much slower. Again, the idea is to run the program within the Delphi IDE, to be able to debug it easily. Notice, though that the list can be expanded with the detailed view, which includes a list of the actual executable files and many other details.


Registered Server Details - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:1024/ServerInfo.ServerInfo
Go
Links
Done
Local intranet

Registered Servers

View List | View Details

COUNTR~1.CountryTutorial
CustQueDebug.CustQueP
DebugDemo.FirstDemo
DebugTest.DebugTest
SERVER~1.ServerInfo
SessionTestTestSvr.session_test
WebSearchDebug.websearchdemo

FIGURE 22.8:

A list of applications registered with the Web App Debugger is displayed when you hook to its *home* page.

The data module for this type of project has some initialization code as well:

```
uses WebReq;

initialization
  WebRequestHandler.WebModuleClass := TWebModule2;
```

This approach should be used only for debugging. To deploy the actual application you should then use one of the other options. What you can do is create the project files for another type of Web server program and add to the project the same Web module of the debug application. The presence of the extra initialization line won't create a problem.

The reverse is slightly more complex. To debug an existing application, you have to create a program of this type, remove the Web module, add the existing one, and patch it by adding a line to set the `WebModuleClass` of the `WebRequestHandler`, like the one in the preceding code snippet. To account for possible missing initialization of the `WebRequestHandler` object, you might want to change this type of code into:

```
if WebRequestHandler <> nil then
  WebRequestHandler.WebModuleClass := ... // Web module class
```

WARNING By doing this for the CustQueP example (it is the CustQueDebug project), I realized that some of the Web request settings are different. So instead of using the `ScriptName` property of the request (set to empty for a Web debug application), you have to use the `InternalScriptName` property.

There are other two interesting elements in the use of the Web App Debugger. The first is that you can test your programs without having a Web server installed and without having to tweak its settings. In other words, you don't have to deploy your programs to test them—you simply try them out right away. Another advantage is that, contrary to doing early development of the applications as CGI, you can start experimenting with a multithreaded architecture right away, without having to deal with the loading and unloading of libraries, which often implies shutting down the Web server and possibly even the computer.

NOTE If your aim is to build an ISAPI application, you can also use a specific ISAPI DLL debugging tool. One such tool, called IntraBob, has been built by Bob Swart and is available on his Web site (`www.drbob42.com`) as freeware.

Working with Apache

If you plan on using Apache instead of IIS or another Web server, you can certainly take advantage of the common CGI technology to deploy your applications on almost any Web server. However, using CGI means some reduced speed and some trouble handling state information (as you cannot keep any data in memory). This is a good reason for writing an ISAPI application or a dynamic Apache module. Using Delphi's WebBroker technology, you can also easily compile the same code for both technologies, so that moving your program to a different Web platform becomes much simpler. Finally, you can also recompile a CGI program or a dynamic Apache module with Kylix and deploy it on a Linux server.

As I've mentioned, Apache can run traditional CGI applications but has also a specific technology for keeping the server extension program loaded in memory at all times for faster response. To build such a program in Delphi 6, you can simply use the Apache Shared Module option of the New Web Server Application dialog box. You end up with a library having this type of source code for its project:

```
library Apache1;

uses
  WebBroker,
  ApacheApp,
  ApacheWm in 'ApacheWm.pas' {WebModule1: TWebModule};
```

```
{$R *.res}

exports
  apache_module name 'apache1_module';

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Notice in particular the `exports` clause, which indicates the name used by Apache configuration files to reference the dynamic module. In the project source code, you can add two more definitions, the module name and the content type, in the following way:

```
ModuleName := 'Apache1_module';
ContentType:= 'Apache1-handler';
```

If you don't set them, Delphi will assign them some default values, which are built adding the *_module* and *-handler* strings to the project name, ending up with the two names I've used above.

An Apache module is generally not deployed within a script folder, but within the modules subfolder of the server itself (by default, `c:\Program Files\Apache\modules`). Then you have to edit the `http.conf` file, adding a line to load the module, as:

```
LoadModule apache1_module modules/apache1.dll
```

Finally, you have to indicate when the module is invoked. The handler defined by the module can be associated with a given file extension (so that your module will process all of the files having a given extension) or with a physical or virtual folder. In the latter case, the folder doesn't exist, but Apache pretends it is there. This is how you can set up a virtual folder for the simple Apache1 module:

```
<Location /Apache1>SetHandler Apache1-handler</Location>
```

As Apache is inherently case sensitive (because of its Linux heritage), you might also want to add a second, lowercase virtual folder:

```
<Location /apache1>SetHandler Apache1-handler</Location>
```

Now you can invoke the sample application with the URL `http://localhost/Apache1`. A great advantage of using virtual folder in Apache is that a user doesn't really distinguish between the physical and dynamic portions of your site, as we'll better see in the next example.

Because the development of Apache modules with WebBroker is almost identical to the development of other types of programs, instead of building an actual application (besides the over-simplistic Apache1 example) I've created a new version of the BrokDemo example, already available as a CGI or ISAPI program. To do this, I've taken the project file of an

Apache module from that example, added the local Web modules to it, and modified the project source code to reflect the proper module name and handler. I've actually defined them differently than the default, as the following code excerpt demonstrates:

```
library BrokApache;

exports apache_module name 'brokdemo_module';

begin
  ContentType:= 'brokdemo-handler';
```

After compiling the module and editing the `http.conf` file as explained above, the program was ready to be used in two different ways, CGI and dynamic module. An obvious difference between the two types of invocation is their URLs:

```
http://localhost/scripts/brokcgi.exe/table
http://localhost/brokdemo/table
```

Not only is the latter URL simpler, but it hides the fact that we are running an application with a `/table` parameter. In fact, it seems we are accessing a specific folder of the server. Actually, the Apache configuration file can be modified to also invoke CGI applications through virtual folders, which explains why CGI applications have a path-like command prefixing the request. Another related explanation is that Linux CGI applications, like any other executable file, have no extension whatsoever, so their names still seem to be part of a path.

Practical Examples

After this general introduction to the core idea of the development of server-side applications with WebBroker, let me end this part of the chapter with two simple practical examples. The first is a classic Web counter. The second is an extension of the WebFind program presented in the preceding chapter to produce a dynamic page instead of filling a list box.

A Web Hit Counter

The server-side applications we've built up to now were based only on text. Of course, you can easily add references to existing graphics files. What's more interesting, however, is to build server-side programs capable of generating graphics that change over time.

A typical example is a *page hit counter*. To write a Web counter, we save the current number of hits to a file and then read and increase the value every time the counter program is called. How do we return this information? If all we need is some HTML text with the number of hits, the code is straightforward:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
```

```
var
  nHit: Integer;
  LogFile: Text;
  LogFileName: string;
begin
  LogFileName := 'WebCont.log';
  System.Assign (LogFile, LogFileName);
  try
    // read if the file exists
    if FileExists (LogFileName) then
    begin
      Reset (LogFile);
      Readln (LogFile, nHit);
      Inc (nHit);
    end
    else
      nHit := 0;
    // saves the new data
    Rewrite (LogFile);
    Writeln (LogFile, nHit);
  finally
    Close (LogFile);
  end;
  Response.Content := IntToStr (nHit);
end;
```

WARNING This simple file handling does not scale. When multiple visitors hit the page at the same time, this code may return false results or fail with a file I/O error because a request in another thread has the file open for reading while this thread tries to open the file for writing. To support a similar scenario, you'll need to use a mutex (or a critical section in a multithreaded program) to let each subsequent thread wait until the thread currently using the file has completed its task.

What's a little more interesting is to create a graphical counter that can be easily embedded into any HTML page. There are basically two approaches for building a graphical counter: you can prepare a bitmap for each digit up front and then combine them in the program, or you can simply let the program draw over a memory bitmap to produce the graphic you want to return. In the WebCount program, I've chosen this second approach.

Basically, we can create an Image component that holds a memory bitmap, which we can paint on with the usual methods of the `TCanvas` class. Then we can attach this bitmap to a `TJpegImage` object. Accessing the bitmap through the JpegImage component converts the image to the JPEG format. At this point, we can save the JPEG data to a stream and return it. As you can see, there are many steps, but the code is not really complex:

```
// create a bitmap in memory
Bitmap := TBitmap.Create;
try
  Bitmap.Width := 120;
  Bitmap.Height := 25;
  // draw the digits
  Bitmap.Canvas.Font.Name := 'Arial';
  Bitmap.Canvas.Font.Size := 14;
  Bitmap.Canvas.Font.Color := RGB (255, 127, 0);
  Bitmap.Canvas.Font.Style := [fsBold];
  Bitmap.Canvas.TextOut (1, 1, 'Hits: ' +
    FormatFloat ('###,###,###', Int (nHit)));
  // convert to JPEG and output
  Jpeg1 := TJpegImage.Create;
  try
    Jpeg1.CompressionQuality := 50;
    Jpeg1.Assign(Bitmap);
    Stream := TMemoryStream.Create;
    Jpeg1.SaveToStream (Stream);
    Stream.Position := 0;
    Response.ContentStream := Stream;
    Response.ContentType := 'image/jpeg';
    Response.SendResponse;
    // the response object will free the stream
  finally
    Jpeg1.Free;
  end;
finally
  Bitmap.Free;
end;
```

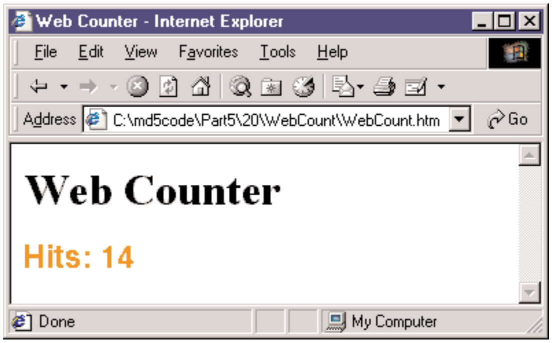
The three statements responsible for returning the JPEG image are the two that set the `ContentStream` and `ContentType` properties of the `Response` and the final call to `SendResponse`. The content type must match one of the possible MIME types accepted by the browser, and the order of these three statements is relevant. There is also a `SendStream` method in the `Response` object, but it should be called only after sending the type of the data with a separate call.

You can see the effect of this program in Figure 22.9. To obtain it, I've added the following code to an HTML page:

```
<img src="http://localhost/scripts/webcount.exe" border=0 alt="hit counter">
```

FIGURE 22.9:

The graphical Web hit counter in action


Web Counter - Internet Explorer
File Edit View Favorites Tools Help
Address C:\md5code\Part5\20\WebCount\WebCount.htm
Go
Done
My Computer

Web Counter

Hits: 14

Searching with a Web Search Engine

In Chapter 21, I discussed the use of the Indy HTTP client component to retrieve the result of a search on the Google Web site. Now I'm going to extend the example a little, turning it into a server-side application. The WebSearch program on the companion CD, available as a CGI application or a Web App Debugger executable, has an action that simply returns the HTML retrieved by the search engine and a second action that fills a client data set component, then hooked to a table page producer. This is the code of this second action:

```
const
  strSearch = 'http://www.google.com/search?as_q=borland+delphi&num=100';

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  I: integer;
begin
  if not cds.Active then
    cds.CreateDataSet
  else
    cds.EmptyDataSet;
  for i := 0 to 5 do // how many pages?
  begin
    // get the data form the search site
    GrabHtml (strSearch + '&start=' + IntToStr (i*100));
    // scan it to fill the cds
    HtmlStringToCds;
  end;
  cds.First;
  // return producer content
  Response.Content := DataSetTableProducer1.Content;
end;
```

The `GrabHtml` method is identical to the WebFind example, while the `HtmlStringToCds` method is similar to corresponding method (which adds the items to a list box) and adds the addresses and their textual descriptions by calling:

```
cds.InsertRecord ([0, strAddr, strText]);
```

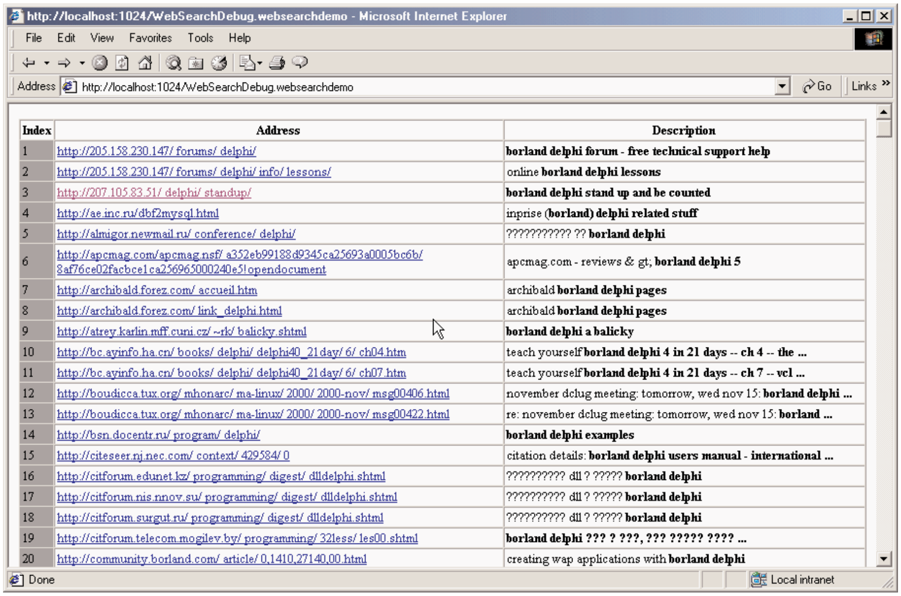
The ClientDataSet component, in fact, is set up with three fields: the two strings plus a line counter. This extra empty field is used to have the extra column in the table producer. The code fills the column in the cell-formatting event, which also adds the hyperlink:

```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if CellRow <> 0 then
  case CellColumn of
    0: CellData := IntToStr (CellRow);
    1: CellData := '<a href="' + CellData + '">' + SplitLong(CellData) + '</a>';
    2: CellData := SplitLong (CellData);
  end;
end;
```

The call to `SplitLong` is used to add some extra spaces within the output text, to avoid having grid columns that are too large, as the browser won't split the text on multiple lines unless it contains spaces or other special characters. The result of this program is a rather slow application (because of the multiple HTTP requests it must forward) producing output like Figure 22.10.

FIGURE 22.10:

The WebSearch program shows the result of the multiple searches done on Google.



http://localhost:1024/WebSearchDebug.websearchdemo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:1024/WebSearchDebug.websearchdemo
Go
Links
Done
Local intranet

Index	Address	Description
1	http://205.158.230.147/ forums/ delphi/	**borland delphi forum - free technical support help**
2	http://205.158.230.147/ forums/ delphi/ info/ lessons/	online **borland delphi lessons**
3	http://207.105.83.51/ delphi/ standup/	**borland delphi stand up and be counted**
4	http://ae.inc.ru/dbf2mysql.html	inprise (**borland) delphi related stuff**
5	http://almigor.newmail.ru/ conference/ delphi/	??????????? ?? **borland delphi**
6	http://apcmag.com/apcmag.nsf/ a352eb99188d9345ca25693a0005bc6b/ 8af76ce02facbce1ca256965000240e5!opendocument	apcmag.com - reviews & gt; **borland delphi 5**
7	http://archibald.forez.com/ accueil.htm	archibald **borland delphi pages**
8	http://archibald.forez.com/ link_delphi.html	archibald **borland delphi pages**
9	http://atrey.karlin.mff.cuni.cz/ ~rk/ balicky.shtml	**borland delphi a balicky**
10	http://bc.ayinfo.ha.cn/ books/ delphi/ delphi40_21day/ 6/ ch04.htm	teach yourself **borland delphi 4 in 21 days -- ch 4 -- the ...**
11	http://bc.ayinfo.ha.cn/ books/ delphi/ delphi40_21day/ 6/ ch07.htm	teach yourself **borland delphi 4 in 21 days -- ch 7 -- vcl ...**
12	http://boudicca.tux.org/ mhonarc/ ma-linux/ 2000/ 2000-nov/ msg00406.html	november dclug meeting: tomorrow, wed nov 15: **borland delphi ...**
13	http://boudicca.tux.org/ mhonarc/ ma-linux/ 2000/ 2000-nov/ msg00422.html	re: november dclug meeting: tomorrow, wed nov 15: **borland ...**
14	http://bsn.docentr.ru/ program/ delphi/	**borland delphi examples**
15	http://citeseer.nj.nec.com/ context/ 429584/ 0	citation details: **borland delphi users manual - international ...**
16	http://citforum.edunet.kz/ programming/ digest/ dlldelphi.shtml	?????????? dll ? ????? **borland delphi**
17	http://citforum.nis.nnov.su/ programming/ digest/ dlldelphi.shtml	?????????? dll ? ????? **borland delphi**
18	http://citforum.surgut.ru/ programming/ digest/ dlldelphi.shtml	?????????? dll ? ????? **borland delphi**
19	http://citforum.telecom.mogilev.by/ programming/ 32less/ 1es00.shtml	**borland delphi ??? ? ???, ??? ????? ???? ...**
20	http://community.borland.com/ article/ 0,1410,27140,00.html	creating wap applications with **borland delphi**

Active Server Pages

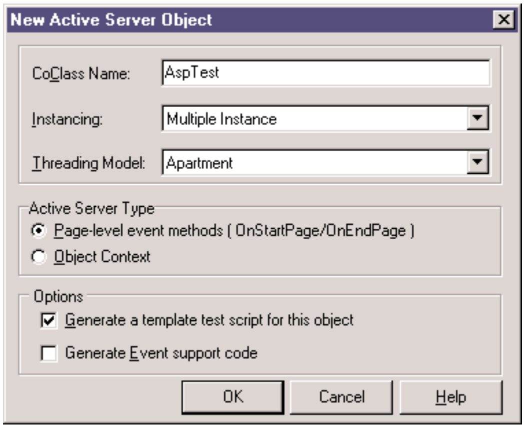
Another approach to the development of server-side applications is the use of scripting. Before looking at the scripting technology embedded in the WebSnap framework, let me shortly discuss Microsoft's Active Server Pages (ASP) technology and how you can use Delphi to support it. The idea behind ASP is to add scripts to the HTML code, so that part of the text on a Web page is directly available while other information can be added at run time on the server. The client receives a plain HTML file. The difference between this approach and ISAPI is that you don't need to recompile a program on the server to see a change; you simply update the script. ASP offers a complex model, where you can attach persistent data to a session (for example, a user moving from page to page of a section of your Web site) and to the entire application (the section of the Web site, regardless of the user).

ASP is quite a complex technology, and here I can only discuss it in relation to Delphi programming. One of the features of ASP is that it allows you to create COM objects within a script, and you can write those COM objects in Delphi. The Delphi IDE even provides specific support classes and a wizard to help you build ASP objects. Compared to ISAPI or CGI, one of the advantages is that your ASP object built in Delphi can get access to session and application information, exactly as an ASP script does. This means we automatically get extra features such as persistent user data built into our server-side object. By building a compiled ASP object, we can also increase the speed of complex server-side code. (ASP scripts are not always the best solution in term of performance.) But, again, I don't want to discuss ASP in detail, only focus on Delphi support.

To try this out, simply create a new ActiveX library, and then start the Active Server Object Wizard (from the ActiveX page of the File ➢ New dialog box). As you can see in Figure 22.11, the wizard has a couple of options. You can build an object integrated with the ASP script by selecting the Page-Level Event Methods radio button, or an internal object (which can be installed as an MTS object) by using the Object Context option. Only in the first case does the object automatically handle the `OnStartPage` method, which receives as parameter the *scripting context*. In both cases, however, the VCL classes you inherit from (`TASPObject` and `TASPMTSObject`, respectively) have properties to access the `Request`, `Response`, `Session`, `Server`, and `Application` ASP objects.

FIGURE 22.11:

The new Active Server Object wizard



New Active Server Object

CoClass Name: AspTest

Instancing: Multiple Instance

Threading Model: Apartment

Active Server Type

● Page-level event methods (OnStartPage/OnEndPage)

○ Object Context

Options

☑ Generate a template test script for this object

☐ Generate Event support code

Once you've created the ASP object with the wizard (I've used the Page-Level Event Methods option for the AspTest example), Delphi will bring up the Type Library editor, where you can prepare a list of properties and methods for your ASP object. Simply add the features you need, and then write their code. For example, you can write the following simple test method:

```
procedure Tasptest.ShowData;
begin
  Response.Write ('<h3>Delphi wrote this text</h3>');
end;
```

and activate it from the following ASP script (only slightly modified from the demo script the Delphi wizard will generate for you):

```
<h4>Message</h4>
<% Set DelphiASPObj = Server.CreateObject("asptest1.asptest")
   DelphiASPObj.showData
%>
```

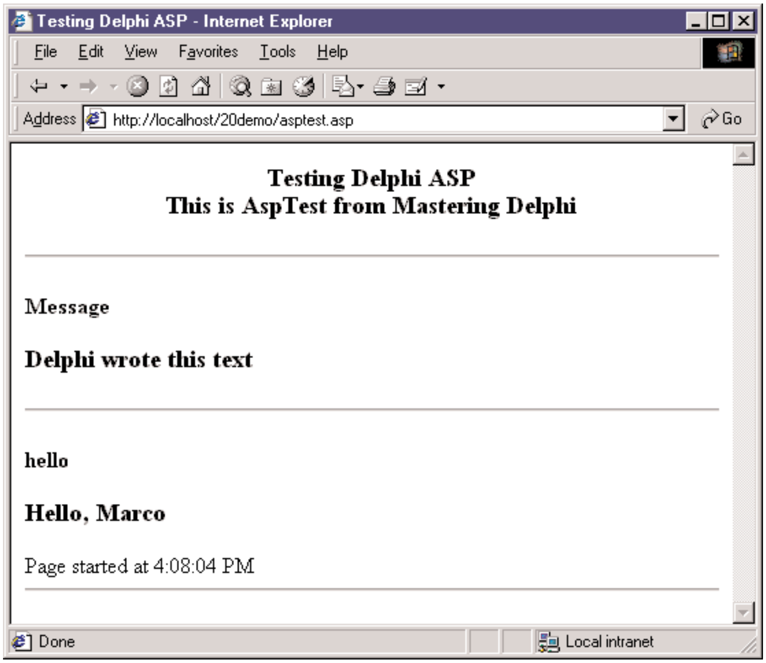
The interesting element is that the same script (or another ASP script of the same application) can also set global values our Delphi object can access. Similarly, multiple objects can communicate, setting global variables for the application and session variables for the specific user. For example, we can add the following text to the ASP page:

```
<h4>hello</h4>
<%
   Session.Value("UserName") = "Marco"
   DelphiASPObj.Hello
 %>
```

I've written the code used to set the property and the method invocation one after the other, but they can even be in different pages. This new *dynamic* property (Microsoft's term for these values added to an object) is saved in the session, so it depends on the current user. The `Hello` method can use the username to welcome them:

```
procedure Tasptest.Hello;
var
  strName: string;
begin
  strName := Session ['UserName'];
  Response.Write ('<h3>Hello, ' + strName + '</h3>');
  Response.Write ('<p>Page started at ' + TimeToStr (StartTime) + '</p>');
end;
```

You can see the result of this and the previous method combined in Figure 22.12. The last line of the method uses a variable that's set when the page is first loaded, in the `OnStartPage` method (despite the name, this is not an event handler, but a method the ASP engine will call as the page containing the object is activated):


Testing Delphi ASP - Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/20demo/asptest.asp
Go
Done
Local intranet

Testing Delphi ASP
This is AspTest from Mastering Delphi

Message

Delphi wrote this text

hello

Hello, Marco

Page started at 4:08:04 PM

FIGURE 22.12: The Web page generated by the AspTest object I've built with Delphi

```
procedure Tasptest.OnStartPage(const AScriptingContext: IUnknown);
begin
  inherited OnStartPage(AScriptingContext);
  StartTime := Now;
end;
```

Technically, this method retrieves the scripting context. The TASPObject base class uses the method to initialize all the ASP objects (including the two, Response and Session, I use in the code), surfacing them as properties.

To generate more complex HTML from the Delphi ASP object, you can use Producer components, optionally connecting them to a dataset. In the AspTest example, I've added a Table component and a DataSetTableProducer, connected them as usual, and written the following code to activate it:

```
procedure Tasptest.ShowTable;
begin
  DataModule1 := TDataModule1.Create (nil);
  try
    Response.Write (DataModule1.DataSetTableProducer1.Content)
  finally
    DataModule1.Free;
  end;
end;
```

It will actually make more sense to create the data module when the COM object is created and destroyed (overriding Initialize and Destroy) or when the page is loaded and unloaded (with OnStartPage and OnEndPage).

WebSnap

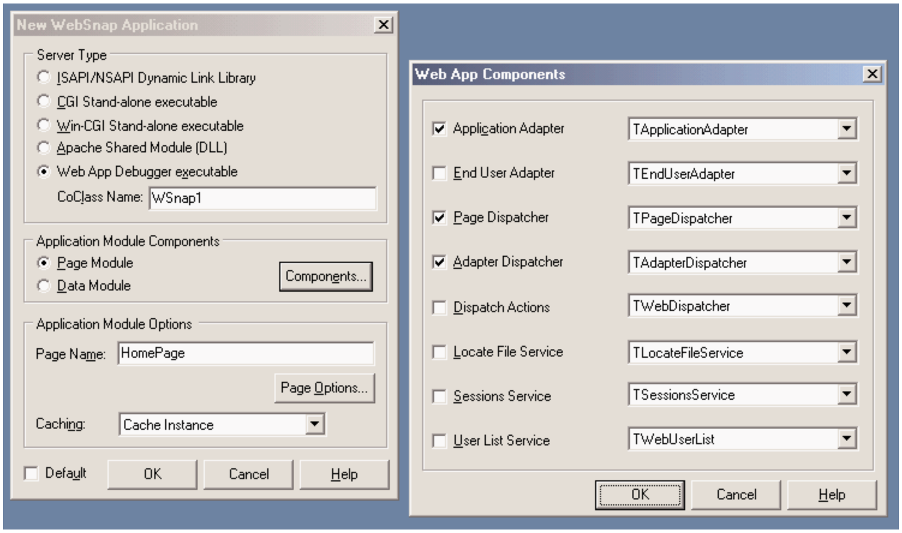
After this lengthy introduction of the core elements of the development of Web server applications with Delphi, we can finally focus on some of the new related technologies introduced in Delphi 6. There were two good reasons for not jumping right into this topic from the beginning of this chapter. The first is that WebSnap builds on the foundation offered by WebBroker, so that you cannot learn how to use the new features if you don't know the core ones. For example, a WebSnap application is technically a CGI or WinCGI program, or an ISAPI or Apache module. The second reason is that since WebSnap is included only in the Enterprise version of Delphi, not all Delphi programmers have the chance to use it (needing to limit their expense to the Professional version of Delphi 6, which includes WebBroker).

WebSnap has a few definitive advantages over the plain WebBroker, such as allowing for multiple pages, integrating server-side scripting, and XSL and Delphi 5 Internet Express technology (these last two elements will be covered in the next chapter). Moreover, there are many ready-to-use components for handling common tasks, such as users' login, session management, and so on. Instead of listing all the features of WebSnap right away, though, I've decided to cover them in a sequence of simple and focused applications. All of these applications have been built using the Web App Debugger, for testing purposes, but you'll be able to easily deploy them using one of the other available technologies.

The starting point of the development of a WebSnap application is a dialog box that you can invoke either in the WebSnap page of the New items dialog box (File ➢ New ➢ Other) or using the new Internet toolbar of the IDE. The resulting dialog box, shown in Figure 22.13, allows you to choose the type of application (like in a WebBroker application) and to customize the initial application components (but you'll be able to add more later on). The bottom portion of the dialog determines the behavior of the first page, usually the default or home page of the program. A similar dialog box is displayed also for subsequent pages.

FIGURE 22.13:

The options offered by the New WebSnap Application dialog box include the type of server and a button for the selection of the core application components.



New WebSnap Application

Server Type

○ ISAPI/NSAPI Dynamic Link Library
○ CGI Stand-alone executable
○ Win-CGI Stand-alone executable
○ Apache Shared Module (DLL)
● Web App Debugger executable

CoClass Name: WSnap1

Application Module Components

● Page Module
○ Data Module

Components...

Application Module Options

Page Name: HomePage

Page Options...

Caching: Cache Instance

□ Default OK Cancel Help

Web App Components

☑ Application Adapter TApplicationAdapter
□ End User Adapter TEndUserAdapter
☑ Page Dispatcher TPageDispatcher
☑ Adapter Dispatcher TAdapterDispatcher
□ Dispatch Actions TWebDispatcher
□ Locate File Service TLocateFileService
□ Sessions Service TSessionsService
□ User List Service TWebUserList

OK Cancel Help

If you go ahead, choosing the defaults and typing in a name for the home page, the dialog box will create a project and open up a `TWebAppPageModule` for you. This module contains the components you've chosen, by default:

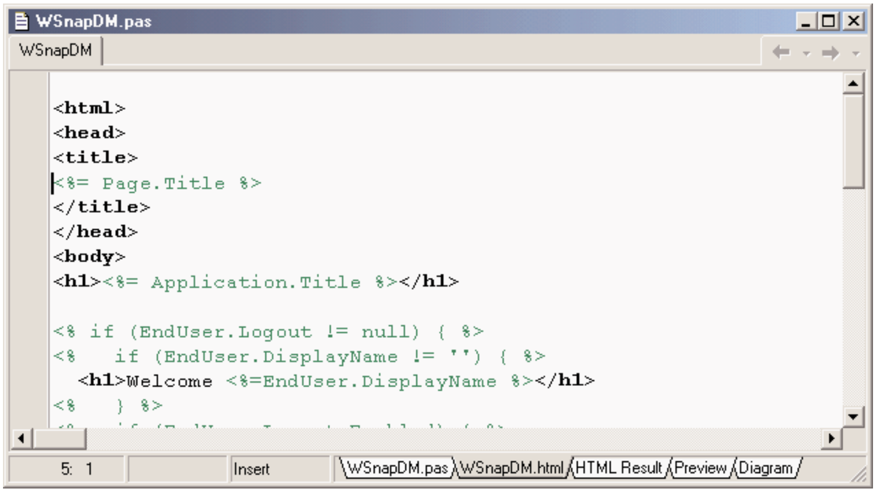
- A WebAppComponents component is a container of all of the centralized services of the WebSnap application, such as the user list, core dispatcher, session services, and so on. Not all of its properties must be available, as an application might not need all of the available services.
- One of these core services is offered by the PageDispatcher component, which (automatically) holds a list of the available pages of the application and defines the default one.
- Another core service is given by the AdapterDispatcher component, which handles HTML form submissions and image requests.
- The ApplicationAdapter is the first component we encounter of the *adapters* family. These components offer fields and actions to the server-side scripts evaluated by the

program. Specifically, the ApplicationAdapter is a fields adapter that exposes the value of its own `ApplicationTitle` property. By entering a value for this property, it will be made available to the scripts.

- Finally, the module hosts a PageProducer that includes the HTML code of the page—in this case, the default page of the program. Unlike WebBroker applications, the HTML for this component is not stored inside its `HTMLDoc` string list property or referenced by its `HTMLFile` property. The HTML file is an external file, stored by default in the folder hosting the source code of the project and referenced from the application using a statement similar to a resource include statement: `{*.html}`.

Because the HTML file included by the PageProducer is kept as a separate file (the LocateFileService component will eventually help you for its deployment), you can edit it to change the output of a page of your program without having to recompile the application. These possible changes relate not only to the fixed portion of the HTML file but also to some of its dynamic content, thanks to the support for server-side scripting. The default HTML file, based on a standard template, actually already has some scripting in it.

The HTML file is visible within the Delphi editor with reasonably good syntax highlighting, simply by selecting the corresponding lower tab, such as WSnapDM.html in my simple example, shown in Figure 22.14. The editor also has other pages for a WebSnap module, including by default an HTML Result page, where you can see the HTML generated after evaluating the scripts, and a Preview page hosting what a user will see inside a browser.



```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1><%= Application.Title %></h1>

<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
```

FIGURE 22.14: The Delphi 6 editor for a WebSnap module includes a simple HTML editor and a preview of its output.

TIP If you prefer editing the HTML of your Web application with another more sophisticated editor, you can set up your choice in the Internet page of the Environment Options dialog box. Within this page, you can see a list of file extensions. Selecting the Edit button for one of these groups of extensions, you can choose an external editor to use for these files. At this point, the External Editor button of the Internet toolbar will become active.

The standard HTML template used by WebSnap adds to any page of the program its title and the application title, using simple script lines such as:

```
<h1><%= Application.Title %></h1>
<h2><%= Page.Title %></h2>
```

We'll get back to the scripting in a while. But let me start the development of the WSnap1 example by simply creating a program with multiple pages. Before I do this, let me finish this overview by showing you the extra source code of a sample Web page module:

```
type
  Thome = class(TWebAppPageModule)
    ...
  end;

function home: Thome;

implementation

{$R *.dfm}  {*.html}

uses WebReq, WebCntxt, WebFact, Variants;

function home: Thome;
begin
  Result := Thome(WebContext.FindModuleClass(Thome));
end;

initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(TWebAppPageModuleFactory.Create(
      Thome, TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html'),
        caCache));
end.
```

The module uses a global function instead of a typical global object of forms to support caching of the pages. This Web App Debugger application also has some extra code in the initialization section, particularly some registration code, to let the application know the role of the page and its behavior.

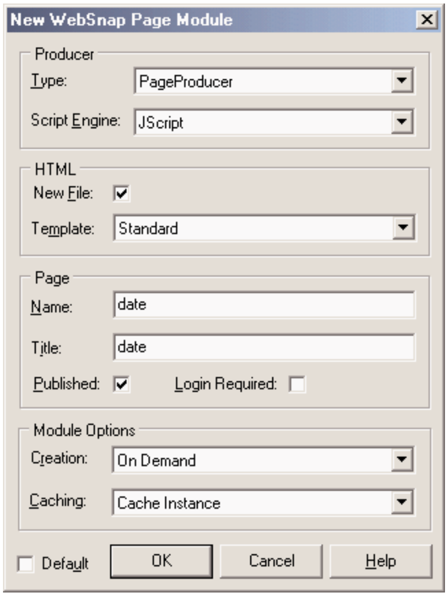
Managing Multiple Pages

The first notable difference between WebSnap and WebBroker is that, instead of having a single data module with multiple actions eventually connected to producer components, WebSnap has multiple data modules, each corresponding to an action and having a producer component with an HTML file attached to it. Actually, you can still add multiple actions to a page/module, but the idea is that you structure applications around pages and not around actions. Like actions, the name of the page is indicated in the request path.

As an example, I've added to the WebSnap application, built with default settings, two more pages. For the first, in the New WebSnap Page Module dialog (see Figure 22.15), I've chosen a standard page producer and given to it the name *date*. For the second, I've gone with a DataSetPageProducer and given it the name *country*. After saving the files, you can start testing the application. Thanks to some of the scripting I'll discuss later, each page lists all of the available pages (unless you've unchecked the Published check box in the New WebSnap Page Module dialog).

FIGURE 22.15:

The New WebSnap Page Module dialog box


New WebSnap Page Module

Producer

Type: PageProducer

Script Engine: JScript

HTML

New File: ☑

Template: Standard

Page

Name: date

Title: date

Published: ☑ Login Required: □

Module Options

Creation: On Demand

Caching: Cache Instance

□ Default

All of the pages will be rather empty, but at least we have the structure in place. To complete the home page, I've simply edited its linked HTML file directly. For the date page, I've employed the same approach as a WebBroker application. I've added to the HTML text some custom tags, as in:

```
<p>The time at this site is <#time>.</p>
```

and I've added some code to the `OnTag` event handler of the producer component to replace this tag with the current time.

For the third page, the country page, I've modified the HTML to include tags for the various fields of the country table, as in:

```
<h3>Country: <#name></h3>
```

Then I've attached the table to the page producer (and I've also added a session component to account for concurrent requests in multiple threads):

```
object DataSetPageProducer: TDataSetPageProducer
  DataSet = Table1
end
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  SessionName = 'Session1_2'
  TableName = 'country.db'
end
object Session1: TSession
  Active = True
  AutoSessionName = True
end
```

To open this table when the page is first created and reset it to the first record in further invocations, I've handled the `OnBeforeDispatchPage` event of the Web page module, adding this code to it:

```
  Table1.Open;
  Table1.First;
```

The fact that a WebSnap page can be very similar to a portion of a WebBroker application (basically an action tied to a producer) is quite important, in case you want to port existing WebBroker code to this new architecture. You can even port your existing DataSetTableProducer components to the new architecture. Technically, you can generate a new page, remove its producer component, replace it with a DataSetTableProducer, and hook this component to the `PageProducer` property of the Web page module. In practice, this approach would cut out the HTML file of the page and its scripts.

In the WSnap1 program, I've used a better technique. I've added a custom tag (`<#htmltable>`) to the HTML file and used the `OnTag` event of the page producer to add to the HTML the result of the data set table:

```
if TagString = 'htmltable' then
  ReplaceText := DataSetTableProducer1.Content;
```

Server-Side Scripts

If having multiple pages in a server-side program, each associated with a different page module, changes the way you write a program, having the server-side scripts at hand offers an even more powerful approach. For example, the standard scripts of the WSnap1 example account for the application and page titles, and for the index of the pages. This is generated by an enumerator, the technique used to scan a list from within a WebSnap script code. Let's have a look at it:

```
<table cellspacing="0" cellpadding="0"><td>
<%  e = new Enumerator(Pages)
    s = ''
    c = 0
    for (; !e.atEnd(); e.moveNext())
    {
      if (e.item().Published)
      {
        if (c > 0) s += '&nbsp;|&nbsp;'
        if (Page.Name != e.item().Name)
          s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
        else
          s += e.item().Title
        c++
      }
    }
    if (c>1) Response.Write(s)
%>
</td></table>
```

NOTE Typically, WebSnap scripts are written in JavaScript, an object-based language very common for Internet programming because it is the only scripting language generally available in browsers (on the client side). JavaScript, technically indicated as ECMAScript, borrows the core syntax of the C language and has almost nothing to do with Java. Actually, WebSnap uses Microsoft's ActiveScripting engine, which supports both JScript (a variation of JavaScript) and VBScript.

Inside the single cell of this table (which, oddly enough, has no rows), the script outputs a string with the `Reponse.Write` command. This string is built with a `for` loop over an enumerator of the pages of the application, stored in the `Pages` global entity. The title of each page is added to the string, only if the page is published and using an hyperlink only for pages different than the current one. Having this code in a script, instead of hard-coded into a Delphi component, allows you to pass it over to a good Web designer to turn it into something a little more visually appealing.

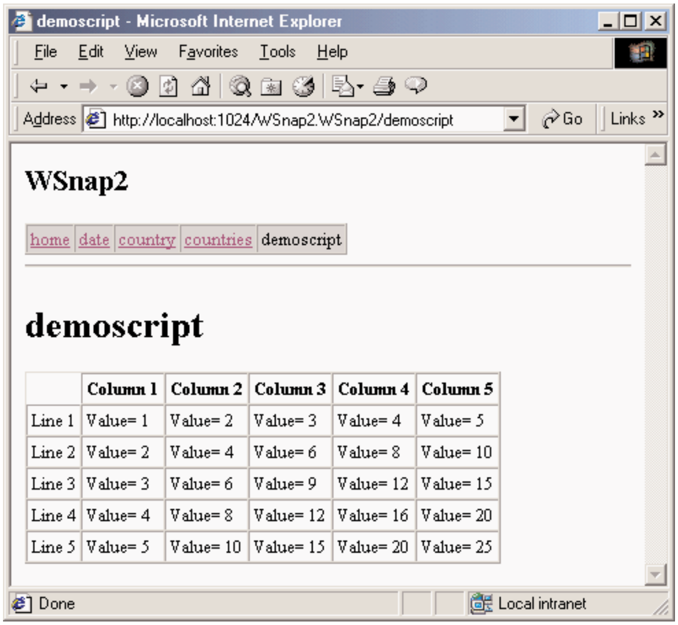
TIP To publish or unpublish a page, don't look for a property in the Web page module. This status is controlled by a flag of the `AddWebModuleFactory` method called in the Web page module initialization code. Simply comment or uncomment this flag to obtain the desired effect.

As a sample of what you can do with scripting, I've added to the WSnap2 example (an extension of the WSnap1 example) a *demoscript* page. The script of this page can generate a full table of multiplied values with the following scripting code (see Figure 22.16 for its output):

```
<table border=1 cellspacing=0>
<tr>
  <th>&nbsp;</th>
  <% for (j=1;j<=5;j++) { %>
  <th>Column <%=j %></th>
  <% } %>
</tr>
<% for (i=1;i<=5;i++) { %>
<tr>
  <td>Line <%=i %></td>
  <% for (j=1;j<=5;j++) { %>
  <td>Value= <%=i*j %></td>
  <% } %>
</tr>
<% } %>
</table>
```

FIGURE 22.16:

The WSnap2 example has a custom menu stored in an included file reference by each page.


demoscript - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:1024/WSnap2.WSnap2/demoscript
Go
Links
Done
Local intranet

WSnap2

home | date | country | countries | demoscript

demoscript

	Column 1	Column 2	Column 3	Column 4	Column 5
Line 1	Value= 1	Value= 2	Value= 3	Value= 4	Value= 5
Line 2	Value= 2	Value= 4	Value= 6	Value= 8	Value= 10
Line 3	Value= 3	Value= 6	Value= 9	Value= 12	Value= 15
Line 4	Value= 4	Value= 8	Value= 12	Value= 16	Value= 20
Line 5	Value= 5	Value= 10	Value= 15	Value= 20	Value= 25

In this script, the <%= symbol replaces the longer `Response.Write` command. Another important feature of server-side scripting is the inclusion of pages within other pages. For example, if you plan on modifying the menu, you can include the related HTML and script in a single file, instead of changing it and maintaining it in multiple pages. File inclusion is generally done with a statement like:

```
<!-- #include file="menu.html" -->
```

In Listing 22.1, you can find the complete source code of the include file for the menu, referenced by all the other HTML files of the project. In Figure 22.16, you can see an example of this menu, across the top of the page with the table generation script mentioned earlier.

Listing 22.1: The *menu.html* file included in each page of the WSnap2 example

```
<html>
<head>
<title><%= Page.Title %></title>
</head>
<body>
<h2><%= Application.Title %></h2>
<table cellspacing="0" cellpadding="2" border="1" bgcolor="#c0c0c0">
<tr>
<%  e = new Enumerator(Pages)
    for (; !e.atEnd(); e.moveNext())
    {
      if (e.item().Published)
      {
        if (Page.Name != e.item().Name)
          Response.Write ('<td><a href="' + e.item().HREF + '">' +
            e.item().Title + '</a></td>')
        else
          Response.Write ('<td>' + e.item().Title + '</td>')
      }
    }
%>
</tr>
</table>
<hr>
<h1><%= Page.Title %></h1>
<p>
```

This script for the menu uses the `Pages` list and the `Page` and `Application` global scripting objects. WebSnap makes available a few other global objects, including `EndUser` and `Session` objects (in case you add the corresponding adapters to the application), the `Modules` object, and the `Producer` object, which allows access to the Producer component of the Web page module. The script also has available the `Response` and `Request` objects of the Web module.

Adapters

Besides these global objects, within a script you can access all the adapters available in the corresponding Web page module. (Adapters in other modules, including shared Web data modules, must be referenced by prefixing their name with the `Modules` object and the corresponding module.) The idea is that adapters allow you to pass information from your compiled Delphi code to the interpreted script, providing a scriptable interface to your Delphi application. Adapters contain fields that represent data and host actions that represent commands. The server-side scripts can access these values and issue these commands, passing specific parameters to them.

NOTE Technically, adapters implement an `IDispatch` interface that can be accessed by the script through an Active Scripting engine language, such as JavaScript. The page producer component is responsible for invoking the Active Scripting engine and has a property indicating the language of the script. Because of this, you'll have to register two type libraries (and deploy the corresponding DLLs) to make this work on a machine where Delphi is not installed: `WebBrokerScript.tlb` and `stdvcl40.dll`. As the first is a type library, it must be installed with Delphi's TRegSvr utility (available in the `bin` subfolder) rather than Microsoft's RegSvr32 program. Of course, the server computer must also have Microsoft Active Scripting Engine installed in order to work.

Adapter Fields

For simple customizations, you can simply add new fields to the specific adapters. For instance, in the WSnap2 example, I've added a custom field to the application adapter. After selecting this component, you can either open up its Fields editor (accessible via its local menu) or simply work within the Object TreeView. After adding a new field (called `Count` in the example), you can assign a value to it in its `OnGetValue` event. As I want to count the hits (or requests) on any page of the Web application, I've also handled the `OnBeforePageDispatch` event of the *global* PageDispatcher component. Here is the code of the two methods:

```
procedure Thome.PageDispatcherBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
begin
  Inc (HitCount);
end;

procedure Thome.CountGetValue(Sender: TObject; var Value: Variant);
begin
  Value := HitCount;
end;
```

Of course, I could have used the page name to also count hits on each specific page (and I could have added some support for persistency, as the count is reset every time you run a new instance of the application). Now that I've added a custom field to an existing adapter (corresponding to the `Application` script object), I can access it from within any script, like this:

```
<p>Application hits since last activation:
<%= Application.Count.Value %></p>
```

Adapter Components

In the same way, you can also add custom adapters to specific pages. If you need to pass along a few fields, use the generic Adapter component. Other custom adapters (besides the global ApplicationAdapter we've already used) include these:

- The PagedAdapter component has built-in support for showing its content over multiple pages.
- The DataSetAdapter component is used to access a Delphi dataset from a script and is covered in the next section.
- The StringValuesList holds a list of name/value pairs, like a string list, and can be used directly or to provide a list of values to an adapter field. The inherited DataSetValuesList adapter has the same role but grabs the list of name/value pairs from a dataset, providing support for lookups and other selections.
- User-related adapters, such as the EndUser, EndUserSession, and LoginForm adapters, are used to access user and session information and to build a login form for the application, automatically tied to the users list. I'll cover these adapters in the section "Sessions, Users, and Permissions" later in this chapter.

Using the AdapterPageProducer

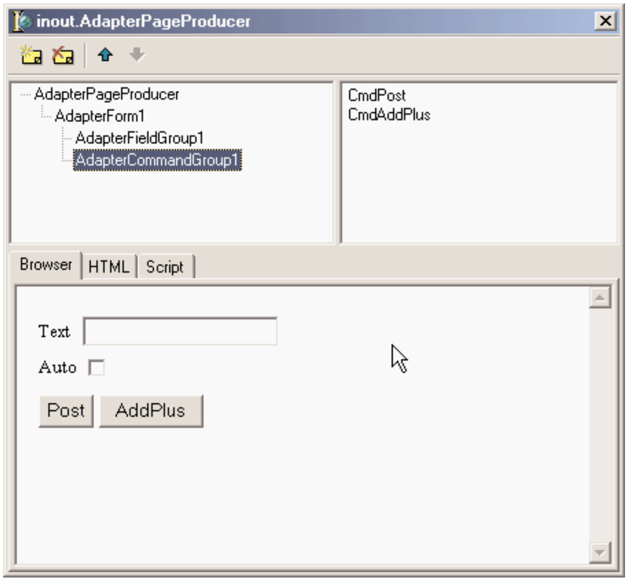
Most of these components are used in conjunction with an AdapterPageProducer component. The AdapterPageProducer, in fact, can generate portions of script after you visually design the desired result. As an example, I've added to the WSnap2 application the *inout* page, which has an adapter with two fields, one standard and one Boolean:

```
object Adapter1: TAdapter
  OnBeforeExecuteAction = Adapter1BeforeExecuteAction
  object TAdapterActions
    object AddPlus: TAdapterAction
      OnExecute = AddPlusExecute
    end
    object Post: TAdapterAction
      OnExecute = PostExecute
    end
  end
  object TAdapterFields
    object Text: TAdapterField
      OnGetValue = TextGetValue
    end
    object Auto: TAdapterBooleanField
      OnGetValue = AutoGetValue
    end
  end
end
```

The adapter has also a couple of actions, used to post the current user input and to add a plus sign to the text. The same plus sign is added anyway when the Auto field is enabled. Developing the user interface for this form, and the related scripting, would take some time using plain HTML. But the AdapterPageProducer component (used in this page) has an integrated HTML designer, which Borland calls Web Surface Designer. Using this tool, you can visually add a form to the HTML page and add an AdapterFieldGroup to it. Connect this field group to the adapter to have editors for the two fields automatically displayed. Then you can add an AdapterCommandGroup and connect it to the AdapterFieldGroup, to have buttons for all of the actions of the adapter. You can see an example of this designer in Figure 22.17.

FIGURE 22.17:

The Web Surface Designer of Delphi 6 for the *inout* page of the WSnap2 example, at design time



To be more precise, the fields and buttons are automatically displayed if the `AddDefaultFields` and `AddDefaultCommands` properties of the field group and command group are set. The effect of the visual operations I've done to build this form are summarized in the following DFM snippet:

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = Adapter1
      object FldText: TAdapterDisplayField
        FieldName = 'Text'
      end
      object FldAuto: TAdapterDisplayField
        FieldName = 'Auto'
      end
    end
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup1
      object CmdPost: TAdapterActionButton
        ActionName = 'Post'
      end
      object CmdAddPlus: TAdapterActionButton
        ActionName = 'AddPlus'
      end
    end
  end
end
```

Now that we have an HTML page with some scripts to move data back and forth and issue commands, we can have a look at the source code required to make this work. First, you'll have to add to the class two local fields to store the adapter fields and manipulate them, and you need to implement the `OnGetValue` event for both, returning the field values. When each of the buttons is pressed, we have to retrieve the text passed by the user, which is not automatically copied into the corresponding adapter field. You can obtain this effect by looking at the `ActionValue` property of these fields, which is set only if something was entered (for this reason, when nothing is entered we set the Boolean field to False). To avoid repeating this code for both actions, I've placed it in the `OnBeforeExecuteAction` event of the Web page module:

```
procedure Tinout.Adapter1BeforeExecuteAction(Sender, Action: TObject;
  Params: TStrings; var Handled: Boolean);
begin
  if Assigned (Text.ActionValue) then
    fText := Text.ActionValue.Values [0];
  fAuto := Assigned (Auto.ActionValue);
end;
```

Notice that each action can have multiple values (in case of components allowing multiple selections); but this is not the case, so we can simply grab the first element. Finally, I've written the code for the OnExecute events of the two actions:

```
procedure Tinout.AddPlusExecute(Sender: TObject; Params: TStrings);
begin
  fText := fText + '+';
end;

procedure Tinout.PostExecute(Sender: TObject; Params: TStrings);
begin
  if fAuto then
    AddPlusExecute (Self, nil);
end;
```

As an alternative, adapter fields have a public EchoActionFieldValue property that you can set to get the value entered by the user and place it again in the resulting form. This technique is typically used in case of errors, to let the user change the input starting with the values already entered.

NOTE The AdapterPageProducer component has specific support for cascading style sheets (CSS). You can define the CSS for a page using either the StylesFile property or Styles string list. Any element of the editor of the items of the producer, at this point, can define a specific style or choose one of the styles of the attached CSS. This last operation (which is the suggested approach) is accomplished using the StyleRule property.

Scripts Rather Than Code?

Even this simple example of the combined use of an adapter and an adapter page producer, with its visual designer, shows the power of this architecture. However, this approach also has a big drawback. By letting the components generate the script (in the HTML, you have only the <#SERVERSCRIPT> tag), you save a lot of development time, but at the same time you end up mixing the script with the code, so that changes to the user interface will require updating the program. The division of responsibilities between the Delphi application developer and the HTML/script designer is lost. And, ironically, we end up having to run a script to accomplish something the Delphi program could have done right away, possibly even much faster!

So my opinion is that this is a very powerful architecture and a huge step forward from WebBroker, but it has to be used with some care, to avoid misusing some of these technologies just because they are simple and powerful (and they are indeed). For example, it might be worth using the designer of the AdapterPageProducer to generate the first version of a page, then grabbing the generated script and copying to the HTML of a plain PageProducer, so that a Web designer can modify the script with a specific tool.

For nontrivial applications, I tend to prefer the possibilities offered by XML and XSL, which are available within this architecture even if they don't have a central role. More on this specific topic in the next chapter.

WebSnap and Databases

One of the areas where Delphi has always shined is database programming. For this reason, it is not surprising to see a lot of support for handling datasets within the WebSnap framework. Specifically, you can use the DataSetAdapter component to connect to a dataset and display its values in a form or a table using the visual editor of the AdapterPageProducer component.

A WebSnap Data Module

As an example, I've built a new WebSnap application (called WSnapTable) with an AdapterPageProducer as its main page to display a table in a grid and another AdapterPageProducer in a secondary page to show a form with a single record. I've also added to the application a WebSnap Data Module, as a container of the dataset components. The data module has a ClientDataSet wired to a dbExpress dataset through a provider and based on an InterBase connection, as shown here:

```
object ClientDataSet1: TClientDataSet
  Active = True
  ProviderName = 'DataSetProvider1'
end
object SQLConnection1: TSQLConnection
  Connected = True
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText =
    'select CUST_NO, CUSTOMER, ADDRESS_LINE1, CITY, STATE_PROVINCE, ' +
    '  COUNTRY from CUSTOMER'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
```

The DataSetAdapter

Now that we have a dataset available, we can add a DataSetAdapter to the first page, and connect it to the ClientDataSet of the Web module. The adapter automatically makes available all of the fields of the dataset and several predefined actions for operating on it (such as

Delete, Edit, and Apply). You can add them explicitly to the `Actions` and `Fields` collections to exclude some of them and customize their behavior, but this is not always required.

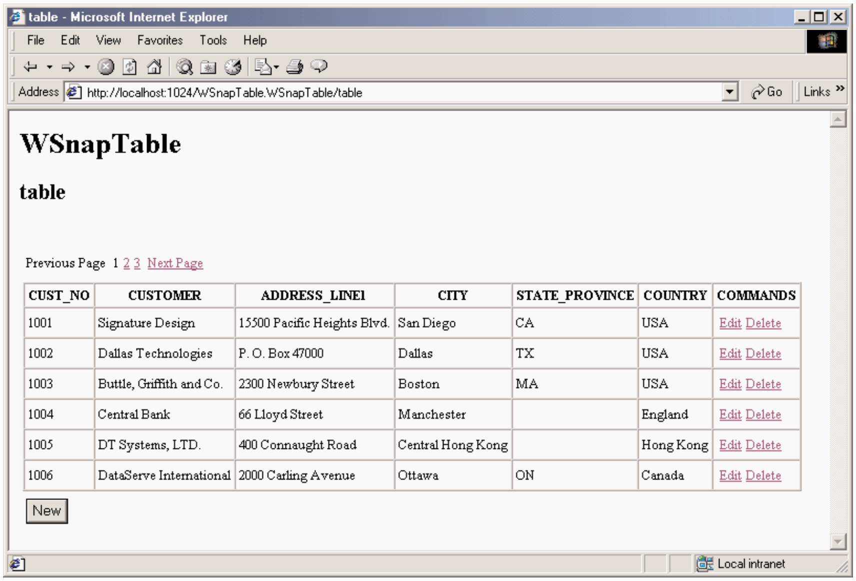
Like the PagedAdapter, the DataSetAdapter has a `PageSize` property where you can indicate the number of elements to display in each page. The component also has commands that you can use to navigate among pages. This approach is particularly suitable when you want to display a large dataset in a grid. These are the adapter settings for the main page of the WSnapTable example:

```
object DataSetAdapter1: TDataSetAdapter
  DataSet = WebDataModule1.ClientDataSet1
  PageSize = 6
end
```

The corresponding page producer has a form containing two command groups and a grid. The first command group (displayed above the gird) has the predefined commands for handling pages: `CmdPrevPage`, `CmdNextPage`, and `CmdGotoPage`. This last command generates a list of numbers for the pages, so that a user can jump to each of them directly. The AdapterGrid component has the default columns plus an extra one hosting a couple of commands, Edit and Delete. The bottom command group has a button used to create a new record. You can see an example of the output of the table in Figure 22.18 and the complete settings of the AdapterPageProducer in Listing 22.2.

FIGURE 22.18:

The page shown by the WSnapTable example at start up includes the initial portion of a *paged* table.



http://localhost:1024/WSnapTable.WSnapTable/table

WSnapTable

table

Previous Page 1 2 3 Next Page

CUST_NO	CUSTOMER	ADDRESS_LINE1	CITY	STATE_PROVINCE	COUNTRY	COMMANDS
1001	Signature Design	15500 Pacific Heights Blvd.	San Diego	CA	USA	Edit Delete
1002	Dallas Technologies	P. O. Box 47000	Dallas	TX	USA	Edit Delete
1003	Buttle, Griffith and Co.	2300 Newbury Street	Boston	MA	USA	Edit Delete
1004	Central Bank	66 Lloyd Street	Manchester		England	Edit Delete
1005	DT Systems, LTD.	400 Connaught Road	Central Hong Kong		Hong Kong	Edit Delete
1006	DataServe International	2000 Carling Avenue	Ottawa	ON	Canada	Edit Delete

New

Listing 22.2: AdapterPageProducer settings for the WSnapTable main page

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
object AdapterCommandGroup1: TAdapterCommandGroup
  DisplayComponent = AdapterGrid1
  object CmdPrevPage: TAdapterActionButton
    ActionName = 'PrevPage'
    Caption = 'Previous Page'
  end
  object CmdGotoPage: TAdapterActionButton
    ActionName = 'GotoPage'
  end
  object CmdNextPage: TAdapterActionButton
    ActionName = 'NextPage'
    Caption = 'Next Page'
  end
end
object AdapterGrid1: TAdapterGrid
  TableAttributes.CellSpacing = 0
  TableAttributes.CellPadding = 3
  Adapter = DataSetAdapter1
  AdapterMode = 'Browse'
  object ColCUST_NO: TAdapterDisplayColumn
    FieldName = 'CUST_NO'
  end
  object ColCUSTOMER: TAdapterDisplayColumn
    FieldName = 'CUSTOMER'
  end
  object ColADDRESS_LINE1: TAdapterDisplayColumn
    FieldName = 'ADDRESS_LINE1'
  end
  object ColCITY: TAdapterDisplayColumn
    FieldName = 'CITY'
  end
  object ColSTATE_PROVINCE: TAdapterDisplayColumn
    FieldName = 'STATE_PROVINCE'
  end
  object ColCOUNTRY: TAdapterDisplayColumn
    FieldName = 'COUNTRY'
  end
  object AdapterCommandColumn1: TAdapterCommandColumn
    Caption = 'COMMANDS'
    object CmdEditRow: TAdapterActionButton
      ActionName = 'EditRow'
      Caption = 'Edit'
      PageName = 'formview'
      DisplayType = ctAnchor
    end
    object CmdDeleteRow: TAdapterActionButton
      ActionName = 'DeleteRow'
      Caption = 'Delete'
      DisplayType = ctAnchor
    end
  end
```

```
        end
        object AdapterCommandGroup2: TAdapterCommandGroup
          DisplayComponent = AdapterGrid1
          object CmdNewRow: TAdapterActionButton
            ActionName = 'NewRow'
            Caption = 'New'
            PageName = 'formview'
          end
        end
      end
    end
```

In this rather long listing, there are a few things to notice. First, the grid has the AdapterMode property set to Browse, other possibilities being Edit, Insert, and Query. This dataset display mode for adapters determines the type of user interface (text or edit boxes and other input controls) and the visibility of other buttons (for example, Apply and Cancel buttons are only present in the edit view, the opposite for the Edit command).

NOTE The adapter mode can also be modified using server-side script and accessing Adapter.Mode.

Second, I've modified the display of the commands inside the grid, using the ctAnchor value for the DisplayType property instead of the default button style. Similar properties are available in most components of this architecture to tweak the HTML code they produce.

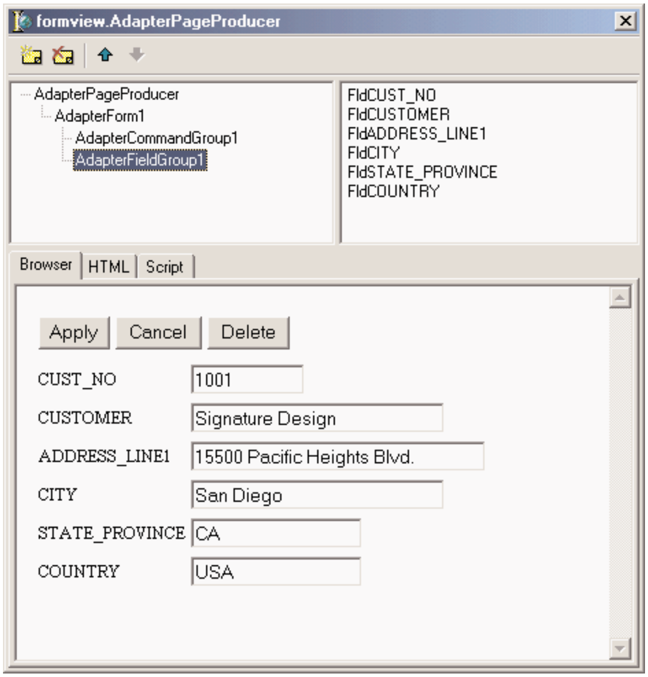
Editing the Data in a Form

Finally, some of the commands are connected to a different page, the page that is going to be displayed after the commands are invoked. For example, the edit command has its PageName property set to formview. This second page of the application has an AdapterPageProducer with components hooked to the same DataSetAdapter of the other table, so that all of the request will be automatically synchronized. Selecting the edit command, in fact, the program will open the secondary page displaying the data of the record corresponding to the command.

Listing 22.3 shows the details of the page producer of the second page of the program. Again, building the HTML form visually using the Delphi specific designer (see Figure 22.19) was a very fast operation.

FIGURE 22.19:

The formview page shown by the WSnapTable example at design time, in the Web Surface Designer (or AdapterPageProducer editor)



CUST_NO	1001
CUSTOMER	Signature Design
ADDRESS_LINE1	15500 Pacific Heights Blvd.
CITY	San Diego
STATE_PROVINCE	CA
COUNTRY	USA

Listing 22.3: AdapterPageProducer settings for the formview page

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterErrorList1: TAdapterErrorList
      Adapter = table.DataSetAdapter1
    end
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup1
      object CmdApply: TAdapterActionButton
        ActionName = 'Apply'
        PageName = 'table'
      end
      object CmdCancel: TAdapterActionButton
        ActionName = 'Cancel'
        PageName = 'table'
      end
      object CmdDeleteRow: TAdapterActionButton
        ActionName = 'DeleteRow'
        Caption = 'Delete'
        PageName = 'table'
      end
    end
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = table.DataSetAdapter1
      AdapterMode = 'Edit'
      object FldCUST_NO: TAdapterDisplayField
        DisplayWidth = 10
        FieldName = 'CUST_NO'
      end
      object FldCUSTOMER: TAdapterDisplayField
        DisplayWidth = 27
        FieldName = 'CUSTOMER'
      end
      object FldADDRESS_LINE1...
      object FldCITY...
      object FldSTATE_PROVINCE...
      object FldCOUNTRY...
    end
  end
end
```

In the listing, you can see that all the operations send the user back the main page and that the AdapterMode is set to Edit, unless there are update errors or conflicts. In this case, the same page is displayed again, with a description of the errors obtained by adding an AdapterErrorList component at the top of the form.

The second page is not published, because selecting it without referring to a specific record would make very little sense. To unpublish the page, I've simply commented the corresponding flag in the initialization code. Finally, to make the changes to the database persistent, you can call the ApplyUdpates method in the OnAfterPost and OnAfterDelete events of the ClientDataSet component hosted by the data module. Another problem (which I haven't fixed) relates to the fact that the SQL server assigns the ID of each customer, so that when you enter a new record, the data in the ClientDataSet and in the actual database are not aligned any more. This can cause Record Not Found errors, a problem I've not fixed in the example.

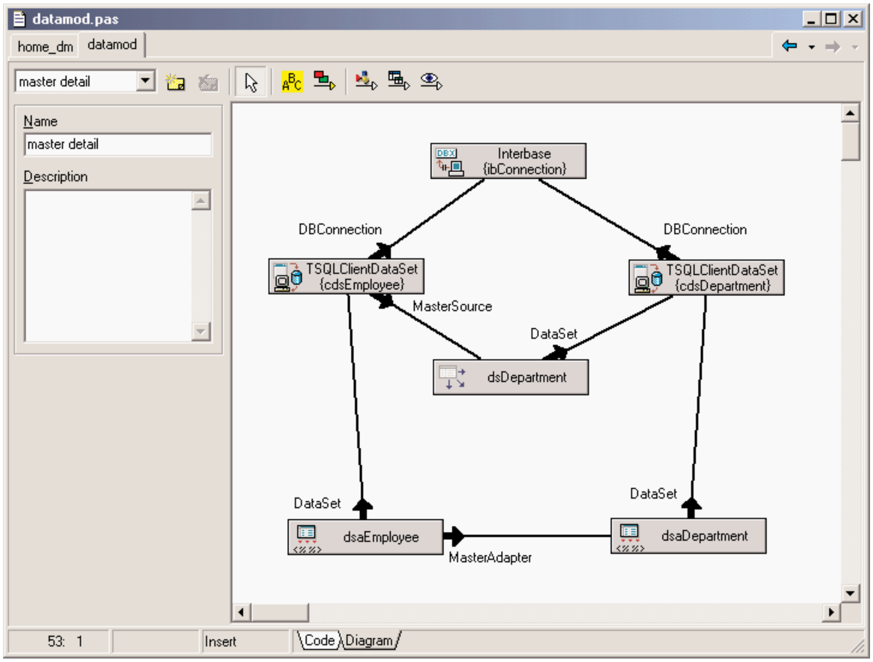
Master/Detail in WebSnap

The DataSetAdapter component has specific support for master/detail relationships between datasets. After you've created the relationship among the datasets, as usual, define an adapter for each dataset and then connect the MasterAdapter property of the adapter of the detail dataset. Setting up the master/detail relationship between the adapters makes them work in a more seamless way. For example, when you change the work mode of the master, or enter new records, the detail automatically enters into Edit mode or is refreshed.

In the WSnapMD example, I've defined such a relationship using two SQLClientDataSet components connected with an InterBase database via dbExpress. All these components and the related adapters are in a Web data module, which has the structure displayed in the design view in Figure 22.20. I haven't provided a complete listing of the details of these components, as it shouldn't be too difficult for you to rebuild it after looking at the example itself.

FIGURE 22.20:

The design view of the Web data module of the WSnapMD example. Both the datasets and the adapters have a master/detail relationship.



Name
master detail

Description

The only page of this WebSnap application has an AdapterPageProducer component hooked to both dataset adapters. The form of this page, in fact, has both a field group hooked to the master and a grid connected with the detail. Unlike other examples, I've tried to improve the user interface by adding custom attributes for the various elements, as you can see in the following detailed excerpt:

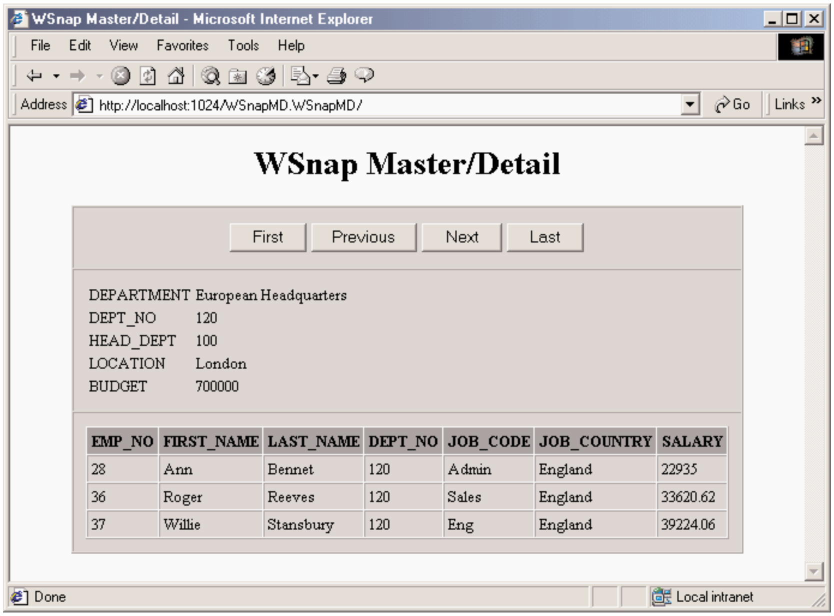
```
object AdapterPageProducer: TAdapterPageProducer
 object AdapterForm1: TAdapterForm
  Custom = 'Border="1" CellSpacing="0" CellPadding="10" ' +
   'BgColor="Silver" align="center"'
  object AdapterCommandGroup1: TAdapterCommandGroup
   DisplayComponent = AdapterFieldGroup1
```

```
      Custom = 'Align="Center"'
      object CmdFirstRow: TAdapterActionButton
        ActionName = 'FirstRow'
        Caption = '   First   '
      end
      object CmdPrevRow: TAdapterActionButton
        ActionName = 'PrevRow'
        Caption = ' Previous '
      end
      object CmdNextRow: TAdapterActionButton
        ActionName = 'NextRow'
        Caption = '   Next   '
      end
      object CmdLastRow: TAdapterActionButton
        ActionName = 'LastRow'
        Caption = '   Last   '
      end
    end
    object AdapterFieldGroup1: TAdapterFieldGroup
      Custom = 'BgColor="Silver"'
      Adapter = WDataMod.dsaDepartment
      AdapterMode = 'Browse'
    end
    object AdapterGrid1: TAdapterGrid
      TableAttributes.BgColor = 'Silver'
      TableAttributes.CellSpacing = 0
      TableAttributes.CellPadding = 3
      HeadingAttributes.BgColor = 'Gray'
      Adapter = WDataMod.dsaEmployee
      AdapterMode = 'Browse'
      object ColEMP_NO: TAdapterDisplayColumn...
      object ColFIRST_NAME: TAdapterDisplayColumn...
      object ColLAST_NAME: TAdapterDisplayColumn...
      object ColDEPT_NO: TAdapterDisplayColumn...
      object ColJOB_CODE: TAdapterDisplayColumn...
      object ColJOB_COUNTRY: TAdapterDisplayColumn...
      object ColSALARY: TAdapterDisplayColumn...
    end
  end
end
```

I've used a gray background, displayed some of the grid borders (HTML grids are used very often by the Web surface designer), centered most of the elements, and added some spacing. Notice that I've added some extra spaces to the button captions, to avoid them being too small. The effect of these settings (and the master/detail structure) is visible at run time in Figure 22.21.

FIGURE 22.21:

The WSnapMD example shows a master/detail structure and has some customized output.



WSnap Master/Detail - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:1024/WSnapMD.WSnapMD/
Go
Links
Done
Local intranet

WSnap Master/Detail

First Previous Next Last

DEPARTMENT European Headquarters
DEPT_NO 120
HEAD_DEPT 100
LOCATION London
BUDGET 700000

EMP_NO	FIRST_NAME	LAST_NAME	DEPT_NO	JOB_CODE	JOB_COUNTRY	SALARY
28	Ann	Bennet	120	Admin	England	22935
36	Roger	Reeves	120	Sales	England	33620.62
37	Willie	Stansbury	120	Eng	England	39224.06

Sessions, Users, and Permissions

Another very interesting area of the WebSnap architecture is its support for sessions and users. Sessions are supported using a classic approach: temporary cookies. These cookies are sent to the browser, so that following requests from the same user can be acknowledged by the system. By adding data to a session instead of an application adapter, you can have data that depends on the specific session or user (although a user can possibly run multiple sessions by opening multiple browser windows on the same computer). For supporting sessions, the application keeps data in memory, so this feature is not available in case of CGI programs.

Using Sessions

To underline the importance of this type of support, I've built a WebSnap application with a single page showing both the total number of hits and the total number of hits for each session. The program has a SessionService component with default values for its `MaxSessions` and `DefaultTimeout` properties. For every new request, the program increases both an `nHits` private field of the page module and the `SessionHits` value of the current session:

```
procedure TSessionDemo.WebAppPageModuleBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
begin
  // increase application and session hits
  Inc (nHits);
  WebContext.Session.Values ['SessionHits'] :=
    Integer (WebContext.Session.Values ['SessionHits']) + 1;
end;
```

NOTE The `WebContext` object (of type `TWebContext`) is a thread variable, created by WebSnap for each request, which provides thread-safe access to other global variables used by program.

The associated HTML displays status information both by using some custom tags evaluated by the `OnTag` event of the page producer and some script, evaluated by the engine. Here is the core portion of the HTML file:

```
<h3>Plain Tags</h3>
<p>Session id: <#SessionID>
<br>Session hits: <#SessionHits></p>
<h3>Script</h3>
<p>Session hits (via application): <%=Application.SessionHits.Value%>
<br>Application hits: <%=Application.Hits.Value%></p>
```

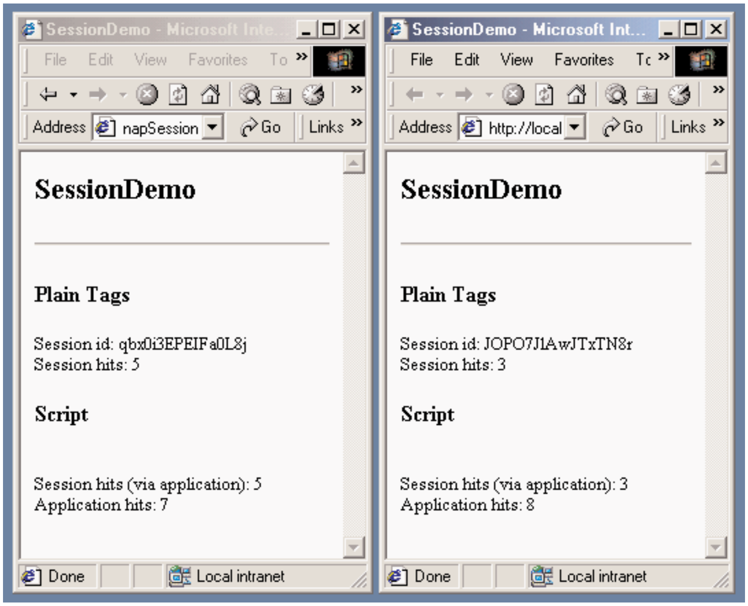
The parameters of the output are provided by the `OnTag` event handler and the `OnGetValue` events of the fields:

```
procedure TSessionDemo.PageProducerHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'SessionID' then
    ReplaceText := WebContext.Session.SessionID
  else if TagString = 'SessionHits' then
    ReplaceText := WebContext.Session.Values ['SessionHits']
end;

procedure TSessionDemo.HitsGetValue(Sender: TObject; var Value: Variant);
begin
  Value := nHits;
end;

procedure TSessionDemo.SessionHitsGetValue(Sender: TObject; var Value: Variant);
begin
  Value := Integer (WebContext.Session.Values ['SessionHits']);
end;
```

The effect of this program is visible in Figure 22.22, where I've activated two sessions in two different copies of Internet Explorer.

FIGURE 22.22:

Two instances of the browser operate on two different sessions of the same WebSnap application.



SessionDemo - Microsoft Inte...
File Edit View Favorites To
Address napSession Go Links
Done Local intranet
SessionDemo - Microsoft Int...
File Edit View Favorites Tc
Address http://local Go Links
Done Local intranet

SessionDemo

Plain Tags

Session id: qbx0i3EPEIFa0L8j
Session hits: 5

Script

Session hits (via application): 5
Application hits: 7

SessionDemo

Plain Tags

Session id: JOPO7J1AwJTxTN8r
Session hits: 3

Script

Session hits (via application): 3
Application hits: 8

TIP In this example, I've voluntarily used both the traditional WebBroker tag replacement and the newer WebSnap adapter fields and scripting, so that you can compare the two approaches and keep in mind that they are both available in a WebSnap application.

Requesting Login

Besides generic sessions, WebSnap also has specific support for users and login-based authorized sessions. You can add to an application a list of users (with the WebUserList component), each with a name and a password. My impression is that this component is rather rudimentary in the data it can store. Instead of filling it with your list of users, however, you can keep the list in a database table (or in some other proprietary format) and use the events of the WebUserList component to retrieve your custom users data and check the user passwords.

You'll generally also add to the application the SessionService and EndUserSessionAdapter components. At this point, you can ask the users to log in, indicating for each page whether it can be viewed by everyone or only by logged-in users. This is accomplished by setting the wpLoginRequired flag in the constructor of the `TWebPageModuleFactory` and `TWebAppPageModuleFactory` classes in the initialization code of the Web page unit.

NOTE The reason for having rights and publication information in the factory rather than in the WebPageModule, is that the program can check the access rights and list the pages even without loading the module.

When a user tries to see a page that requires the user identification, the login page indicated in the EndUserSessionAdapter component is displayed. You can create such a page rather easily by creating a new Web page module based on an AdapterPageProducer and adding to it the LoginFormAdapter. In the editor of the page, add a field group within a form, connect the field group to the LoginFormAdapter, and add a command group with the default Login button. The resulting login form will have fields for the username and its password, but also for the requested page. This last value is automatically filled with the requested page, in case this page required authorization and the user wasn't already logged in. This is done so that a user can immediately reach the requested page without being bounced back to a generic menu.

The login form is typically not published, because the corresponding Login command is already available when the user isn't logged into the system; when the user logs in, it is replaced by a Logout command. This is obtained by the standard script of the Web Page Module, and particularly:

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
<%   if (EndUser.Logout.Enabled) { %>
  <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
  <a href=<%=EndUser.LoginForm.AsHREF%>>Login</a>
<%   } %>
<% } %>
```

There isn't much else to say about the WSnapUsers application, as it has almost no custom code and settings. The access to the users data is demonstrated by the script of the standard template shown above.

Single Page Access Rights

Besides having pages that require a login for access, you can give specific users the right to see more pages than others. Any user, in fact, has a set of rights separated by semicolons or commas. The user must have all of the rights defined for the requested page generally listed in the `ViewAccess` and `ModifyAccess` properties of the adapters, which indicate respectively whether the user can see the given elements while browsing or can even edit them. These settings are very granular, and can be applied to entire adapters or some specific adapter fields (notice I'm referring to the adapter fields, not the user interface components within the designer). For example, you can hide some of the columns of a table to given users by hiding the corresponding fields (and also in other cases, as specified by the `HideOptions` property).

The global PageDispatcher component also has the `OnCanViewPage` and `OnPageAccessDenied` events that can also be used to control the access to the various pages of the program within the program code, allowing for even greater control.

What's Next?

In this chapter, I've covered Web server applications, using multiple techniques (CGI, ISAPI, Apache dynamic modules) and two different frameworks of the Delphi class library: WebBroker and WebSnap. This wasn't certainly an in-depth presentation, as one could write an entire book on this topic alone. It was intended as a starting point, and (as usual) I've tried to make the core concepts clear rather than building very complex examples.

If you want to learn more details of the WebSnap framework and see different examples in actions, refer to the extensive Delphi demos for this area, in the `\Demos\WebSnap` folder. Some of the other available options, relating to XML, XSL, and client-side scripts, will be examined in the next chapter, where I'll also discuss Web services as a powerful alternative to HTTP/HTML-based distributed applications.

CHAPTER 23

XML and SOAP

- Introducing XML, Extensible Markup Language
- Working with XML: DOM and SAX
- Delphi 6 and XML: interfaces and mapping
- Internet Express
- Using XSTL
- Web services
- SOAP and WSDL

Building applications for the Internet means using protocols and creating browser-based user interfaces, as we've done in the preceding two chapters, but also opens up the opportunity of exchanging business documents electronically. The emerging standards for this type of activity all center around the XML document format and include the SOAP transmission protocol, XML schemas for the validation of documents, and XSL for rendering them as HTML.

In this chapter, I'll discuss all of these technologies and the extensive support Delphi 6 offers for them, a series of features collectively known as BizSnap. Since you might not know XML and related technologies, I'll provide a little introduction about each of them, but you should refer to books specifically devoted to each subject to know more. What I won't try to do is to cover *why* this is a revolution for running the IT side of a business and what it opens up. In the conclusion of the chapter, I'll point you to some initiatives you might be interested in tracking.

Introducing XML

XML, or Extensible Markup Language, is a simplified version of SGML and is getting a lot of attention in the IT world. XML is a *markup language*, meaning it uses symbols to describe its own content—in this case, *tags* consisting of specially defined text enclosed in angle brackets. It is named *extensible* because it allows for free markers (in contrast, for example, to HTML, which has predefined markers). The XML language is a standard promoted by the World Wide Web Consortium (better known as W3C, `www.w3.org`).

TIP The XML Recommendation is at `www.w3.org/TR/REC-xml`.

XML has been touted as the ASCII of year 2000, to indicate a simple and widespread technology, and also to indicate that XML document is actually a plain text file (optionally with Unicode characters instead of plain ACSII text). The important element of XML is that it is descriptive, as every tag has an almost human-readable name. Here is a small example, in case you've never seen an XML document:

```
<book>
  <title>Mastering Delphi 6</title>
  <author>Cantu</author>
  <publisher>Sybex</publisher>
</book>
```

WARNING XML has also a few disadvantages I want to underline from the beginning. The biggest is that without a formal description, a document is worth very little. If you want to exchange documents with another company, you have to agree on what each tag means and also on the semantic meaning of the content. (For example, when you have a quantity, you have to agree on the measurement system or include it in the document.) Another disadvantage is that XML documents are much larger than other formats; using strings for numbers, for example, is far from efficient, and the repeated opening and closing tags eat up a lot of space. The good news is that XML compresses quite well, exactly for the same reasons.

Core XML Syntax

There are a few technical elements of XML that are worth knowing before discussing its usage within Delphi. Here is a short summary of the key elements of the XML syntax:

- White space (including the space character, carriage return, line feed, and tabs) is generally ignored (as in an HTML document). It is important to format an XML document to make it readable by a human being, but your programs won't care much.
- You can add comments within `<!--` and `-->` markers, which are basically ignored by any XML processor. There are also directives and processing instructions, enclosed within the `<?` and `?>` markers.
- There a few special or reserved characters you cannot use in the text. The only two symbols you can *never* use are the less-than character (or "left angle bracket," used to delimit a marker) replaced by `&lt;` and the ampersand character replaced by `&amp;`. Other optional special characters are `&gt;` for the greater-than symbol (right angle bracket), `&apos;` for the single quote, and `&quot;` for the double quote.
- To add non-XML content (for example, binary information or a script), you can use a CDATA section, enclosed within `<![CDATA[` and `]]>`.
- All markers are enclosed by angle brackets, < and >. Markers are case sensitive (in contrast to HTML).
- For each opening marker, you must have a matching closing marker, denoted by an initial slash character, as in:

  ```
  <node>value</node>
  ```

- Markers must not overlap—they must be *properly nested*, as in the first line below (the second line is not correct):

  ```
  <node>xx <nested> yy</nested> </node>  // OK
  <node>xx <nested> yy</node> </nested>  // WRONG
  ```

- If a marker has no content (but its presence is important anyway), you can replace the opening and closing markers with an single marker that includes a final or "trailing" slash: `<node/>`.
- Markers can also have attributes, using multiple attribute names followed by a value enclosed within quotes: `<node attrib1="aaa">`.
- Any XML node can have multiple attributes, multiple embedded tags, and only one block of text, representing the value of the node. If it's technically possible, it is common practice for XML nodes to have either a textual value or embedded tags, and not both. Here is an example of the full syntax of a node:

  ```
  <node attrib1="aaa" attrib2="bbb">
    value1
    <child1>
      value2
    </child1>
  </node>
  ```

- A node can have multiple child nodes with the same tag (tags need not be unique). Attribute names are unique for each node.

Well-Formed XML

If the elements discussed in the previous section define the syntax of an XML document, they are not enough. A XML document is considered syntactically correct, or *well formed*, if it follows a few extra rules. Notice that this type of check doesn't guarantee that the content of the document is meaningful, but only the tags are properly laid out.

One of the rules is that each document should have a prologue, indicating that is it indeed an XML document, which version of XML it complies with, and the possibly the type of character encoding. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Possible encodings include various Unicode character sets (such as UTF-8, UTF-16, and UTF-32) or some ISO encodings (such as ISO-10646-xxx or ISO-8859-xxx). The prologue can also include external declarations, the schema used to validate the document, namespace declarations, an associated XSL file, and some internal entity declarations. Refer to XML documentation or books for more information on these topics.

An XML document is well formed if it has a prologue, has a proper syntax (see the rules in the previous section), and has a tree of nodes with a single root. Most tools (including Internet Explorer) check whether a document is well formed when loading it.

NOTE As you can see, XML is more formal and precise than HTML. The W3C is coming along with an XHTML standard that will make HTML documents XML-compliant, for better processing with XML tools. This implies many changes in a typical HTML document, such as avoiding attributes with no values, adding all the closing markers (as `</p>` and `</li>`), adding the slash to stand-alone markers (as `<hr/>` and `<br/>`), proper nesting, and more. An HTML-to-XHTML converter, called HTML Tidy, is hosted by the W3C Web site at `www.w3.org/People/Raggett/tidy/`.

Working with XML

To get acquainted with the format of XML, you can use one of the existing XML editors available on the market. You can also simply type your XML code into Notepad and then try to load it into Internet Explorer to see whether it is correct. In this case, you'll see it within the browser in a tree-like structure.

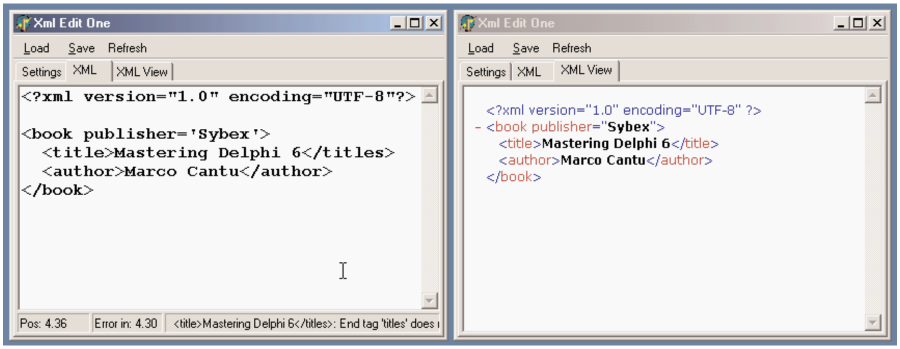
To speed up this type of operation, I've build the simplest XML editor I could come up with—basically Notepad with some XML syntax-checking and a browser attached to it. The XmlEditOne example has a PageControl with three pages. The first page, Settings, hosts a couple of components where you can insert the path and the name of the file and you want to work with. (The reason for not using a standard dialog will become clear when I show you an extension of the program.) The edit box hosting the complete filename is automatically updated with the path and filename, provided the AutoUpdate check box is selected.

The second page hosts a Memo control, with the text of the XML file, loaded and saved by clicking the two toolbar buttons. As soon as you load the file, or each time you modify its text, its content is loaded into a DOM to let a parser check for its correctness (something that would be quite complex to do with your own code). To parse the code, I've used the XMLDocument component available in Delphi 6, which is basically a wrapper around a DOM available on the computer and indicated by its `DOMVendor` property. I'll discuss the use of this component in a little more detail in the next section. For the moment, suffice to say you can assign a string list to its `XML` property and activate it to let it parse the XML text and eventually report an error with an exception.

For this specific example, though, this behavior is far from good, because while typing the XML code you'll have temporarily incorrect XML. Still, I prefer not to ask the user to click a button to do the validation, but let it run continuously. As it is not possible to disable the parse exception raised by the XMLDocument component, I had to work at a lower level, extracting the `DOMPersist` property (referring to the persistency interface of the DOM) after extracting the `IXMLDocumentAccess` interface from the XMLDocument component (called `XmlDoc` in this code). At this point, I can also extract the IDOMParseError interface from the document component, to display any error message in the status bar:

```
procedure TFormXmlEdit.MemoXmlChange(Sender: TObject);
var
  eParse: IDOMParseError;
begin
  XmlDoc.Active := True;
  xmlBar.Panels[1].Text := 'OK';
  xmlBar.Panels[2].Text := '';
  (XmlDoc as IXMLDocumentAccess).DOMPersist.loadxml(MemoXml.Text);
  eParse := (XmlDoc.DOMDocument as IDOMParseError);
  if eParse.errorCode <> 0 then
    with eParse do
    begin
      xmlBar.Panels[1].Text := 'Error in: ' + IntToStr (Line) + '.' +
        IntToStr (LinePos);
      xmlBar.Panels[2].Text := SrcText + ': ' + Reason;
    end;
end;
```

You can see an example of the output of the program in Figure 23.1, alongside the XML tree view provided by the third page (for a correct document). The third page of the program is built using the WebBrowser component, which embeds the ActiveX control of Internet Explorer. Unfortunately, there is no direct way to assign a string with the XML text to this control, so you'll have to save the file first and then move to its page to trigger the loading of the XML in the browser (or manually click the Refresh button).



```
<?xml version="1.0" encoding="UTF-8"?>

<book publisher='Sybex'>
  <title>Mastering Delphi 6</titles>
  <author>Marco Cantu</author>
</book>
```

```
Pos: 4.36 | Error in: 4.30 | <title>Mastering Delphi 6</titles>: End tag 'titles' does
```

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <book publisher="Sybex">
    <title>Mastering Delphi 6</title>
    <author>Marco Cantu</author>
  </book>
```

FIGURE 23.1:

The XmlEditOne example allows you to enter XML text in a memo, indicating errors as you type, and shows the result in the embedded browser.

WARNING Unicode support in Internet Explorer 5 is quite limited, even on Windows 2000 (which has, in general, rather good Unicode support). If you change the final letter in my last name to an accented letter, as it should be, you won't see any problem when checking the document with the DOM, but you'll see an error when viewing it in the browser. The same happens if you change the format to UTF-16.

Managing XML Documents

Now that you know the core elements of XML, we can start discussing how to manage XML documents in Delphi programs (or in programs in general, as some of the techniques discussed here go beyond the language used). There are two typical techniques for manipulating XML documents, using a Document Object Model (DOM) interface or using the Simple API for XML (SAX). The two approaches are quite different:

- The DOM loads the entire document into a hierarchical tree of nodes, allowing you to read them and manipulate them to change the document. For this reason, the DOM is suited when you want to navigate the XML structure in memory and edit it, or even for creating new documents from scratch.
- The SAX parses the document firing an event for each element of the document, without building any structure in memory. Once parsed by the SAX, the document is lost, but this operation is generally much faster than the construction of the DOM tree. Using the SAX is good for reading a document once, possibly looking for portion of its data.

There is a third classic way to manipulate (and specifically create) XML documents: string management. Creating a document by adding strings is certainly the fastest operation, particularly if you can do a single pass (and don't need to modify nodes already generated). Even reading documents by means of string functions is very fast, but this can become quite difficult for complex structures.

Finally, Delphi 6 provides two more techniques you should consider. The first is the definition of interfaces mapping the document structure and used to access the document instead of the generic DOM interface. As we'll see, this approach makes for faster coding and more robust applications. Another technique is the development of transformations that allow you to read a generic XML document into a ClientDataSet component or save the dataset into an XML file of a given structure (not the specific XML structure natively supported by the ClientDataSet, or MyBase).

I won't try to fully assess which option is better suited for each type of document and manipulation, but I will try to highlight some of the advantages and disadvantages while discussing examples of each approach in the next sections.

Programming with the DOM

Since an XML document has a tree-like structure, loading an XML document into a tree in memory is quite a natural fit. This is what the Document Object Model does. The DOM is a standard interface, so that when you have written code that uses a DOM, you can switch DOM implementations without changing your source code (at least, if you haven't used any non-custom extensions).

In Delphi, you can install several DOM implementations, available as COM servers, and use their interfaces. One of the most commonly used DOMs on Windows is the one provided by Microsoft as part of the MSXML SDK. That DOM is also used by Internet Explorer (even if generally in an older version), but the SDK contains some rather detailed documentation, which will probably help you. Other frequently used DOMs are available from IBM and Apache (this one is called Xerces).

TIP There are also a couple of native Object Pascal DOM components. One is the open source OpenXML, available at `www.philo.de/xml`. Another native Delphi DOM is offered by TurboPower. The advantage of these solutions is that they don't require an external library for the program to execute, because the DOM component gets compiled into your application.

Delphi 6 embeds the DOM implementations into a wrapper component, called XMLDocument. I've just used this component in the preceding example, but here I want to examine its role in a more general way. The idea behind using this component is that, instead of the actual DOM interface, you remain even more independent from the implementations and can work with some simplified methods, or helpers.

The DOM interface, in fact, is quite complex to use. A document is a collection of nodes, each having a name, a text element, a collection of attributes, and a collection of child nodes. Each collection of nodes allows accessing elements by position or searching them by name. Notice that the text within the tags of a node, if any, is rendered as a child of the node and listed in its collection of child nodes. The root node has some extra methods for creating new nodes, values, or attributes.

With Delphi's XMLDocument, you can actually work at two different levels:

- At a lower level, you can use the `DOMDocument` property (of the `IDOMDocument` interface type) to access a standard W3C Document Object Model interface. The official DOM is defined in the xmldom unit, and includes interfaces like `IDOMNode`, `IDOMNodeList`, IDOMAttr, IDOMElement, and IDOMText. With the official DOM interfaces, Delphi supports a lower-level but standard programming model. Notice that the actual DOM implementation will be the one indicated by the XMLDocument component.

- As a higher-level alternative, the XMLDocument component implements also the IXMLDocument interface. This is a custom DOM-like interface defined by Borland in the XMLIntf unit and comprising interfaces like IXMLNode, IXMLNodeList, and IXMLNodeCollection. This Borland interface simplifies some of the DOM operations by replacing multiple method calls, which are repeated quite often in sequence, with a single property or method.

In the following examples (particularly the DomCreate demo), I'll use both approaches so you can have a better idea of the practical differences among the two approaches.

An XML Document in a TreeView

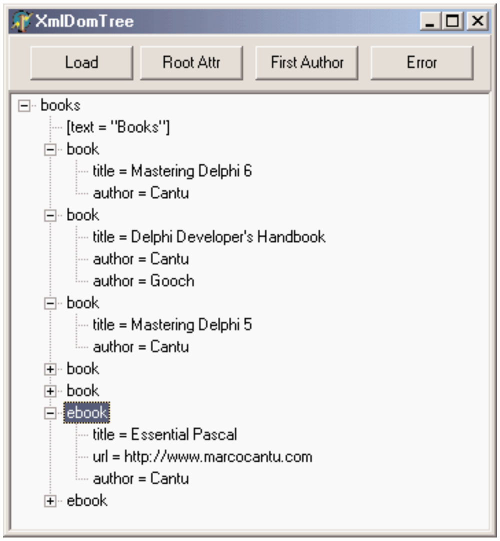
The starting point is generally loading a document from a file or creating it from a string, but you can also start with a brand new document. As a first example of the use of the DOM, I've built a program that can load an XML document into a DOM and show its structure in a TreeView control. I've also added to the program, called XmlDomTree on the companion CD, a few buttons with sample code used to access to the elements of a sample file, as an example of accessing the DOM data. Loading the document is actually quite simple, while showing it in a tree requires a recursive function that navigates the nodes and subnodes. Here is the code of the two methods:

```
procedure TFormXmlTree.btnLoadClick(Sender: TObject);
begin
  OpenDialog1.InitialDir := ExtractFilePath (Application.ExeName);
  if OpenDialog1.Execute then
  begin
    XMLDocument1.LoadFromFile(OpenDialog1.FileName);
    Treeview1.Items.Clear;
    DomToTree (XMLDocument1.DocumentElement, nil);
    TreeView1.FullExpand;
  end;
end;

procedure TFormXmlTree.DomToTree (XmlNode: IXMLNode; TreeNode: TTreeNode);
var
  I: Integer;
  NewTreeNode: TTreeNode;
  NodeText: string;
  AttrNode: IXMLNode;
begin
  // skip text nodes and other special cases
  if not (XmlNode.NodeType = ntElement) then
    Exit;
  // add the node itself
  NodeText := XmlNode.NodeName;
  if XmlNode.IsTextElement then
    NodeText := NodeText + ' = ' + XmlNode.NodeValue;
  NewTreeNode := TreeView1.Items.AddChild(TreeNode, NodeText);
  // add attributes
  for I := 0 to xmlNode.AttributeNodes.Count - 1 do
  begin
    AttrNode := xmlNode.AttributeNodes.Nodes[I];
    TreeView1.Items.AddChild(NewTreeNode,
      '[' + AttrNode.NodeName + ' = "' + AttrNode.Text + '"]');
  end;
  // add each child node
  if XmlNode.HasChildNodes then
    for I := 0 to xmlNode.ChildNodes.Count - 1 do
      DomToTree (xmlNode.ChildNodes.Nodes [I], NewTreeNode);
end;
```

This code is quite interesting, as it highlights some of the operations you can do with a DOM. First of all, each node has a `NodeType` property that you can use to determine whether the node is an element, attribute, text node, or special entity (such as CDATA and others). Another aspect is that you cannot access the textual representation of the node, its `NodeValue`, unless it has a text element (notice that the text node will be skipped, as per the initial test). After displaying the name of the item, and then the text value if available, the program (Figure 23.2) shows the content of each attribute directly and of each subnode calling the `DomToTree` method recursively.

FIGURE 23.2:

The XmlDomTree example can open a generic XML document and show it inside a TreeView common control.



```
⊟ books
    [text = "Books"]
  ⊟ book
      title = Mastering Delphi 6
      author = Cantu
  ⊟ book
      title = Delphi Developer's Handbook
      author = Cantu
      author = Gooch
  ⊟ book
      title = Mastering Delphi 5
      author = Cantu
  ⊞ book
  ⊞ book
  ⊟ ebook
      title = Essential Pascal
      url = http://www.marcocantu.com
      author = Cantu
  ⊞ ebook
```

Once you have loaded the sample document that accompanies the XmlDomTree program (and shown in Listing 23.1) into the XMLDocument component, you can use the various methods to access generic nodes, as in tree-building code above, or fetch specific elements. For example, you can grab the value of the attribute *text* of the root node by writing:

```
XMLDocument1.DocumentElement.Attributes ['text']
```

Notice that if there is no attribute called *text*, the call will fail with a rather generic error message, "Invalid variant type conversion," which helps neither you nor the end user to understand what's wrong. If you need to access to the first attribute of the root without knowing its name, you can use the following more generic code:

```
XMLDocument1.DocumentElement.AttributeNodes.Nodes[0].NodeValue
```

To access the actual nodes, you use a similar technique, possibly taking advantage of the `ChildValues` array. This is a Delphi extension to the DOM, which allows you to pass as parameter either the name of the element or its numeric position:

```
XMLDocument1.DocumentElement.ChildNodes.Nodes[1].ChildValues['author']
```

This code gets the (first) author of the second book. I cannot use the `ChildValues['book']` expression, as there are multiple nodes with the same name under the root node.

Listing 23.1: The sample XML document used by examples in this chapter

```
<?xml version="1.0" encoding="UTF-8"?>
<books text="Books">
  <book>
    <title>Mastering Delphi 6</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi Developer's Handbook</title>
    <author>Cantu</author>
    <author>Gooch</author>
  </book>
  <book>
    <title>Mastering Delphi 5</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi COM Programming</title>
    <author>Harmon</author>
  </book>
  <book>
    <title>Thinking in C++</title>
    <author>Eckel</author>
  </book>
  <ebook>
    <title>Essential Pascal</title>
    <url>http://www.marcocantu.com</url>
    <author>Cantu</author>
  </ebook>
  <ebook>
    <title>Thinking in Java</title>
    <url>http://www.mindview.com</url>
    <author>Eckel</author>
  </ebook>
</books>
```

Creating Documents Using the DOM

Although I mentioned earlier that you can create an XML document by chaining together some strings, this is far from a robust technique. Using a DOM to create a document ensures that the XML will be well formed. Also, if the DOM has a schema definition attached, you can validate the structure of the document while adding data to it.

To highlight different cases of document creation, I've built the DomCreate example (a program I originally created with the Xerces DOM from the Apache group and changed slightly for supporting Delphi's XMLDocument). This program can create XML documents within the DOM, showing the text of them on a memo and optionally in a TreeView.

TIP The XMLDocument component uses the `doAutoIndent` option to improve the output of the XML text to the memo by formatting the XML in a slightly better way. You can choose the type of indentation by setting the `NodeIndentStr` property. For formatting a generic XML text, you can also use the global `FormatXMLData` function using the default setting (2 spaces) as indentation. Oddly, there doesn't seem a way to pass a different parameter to the function.

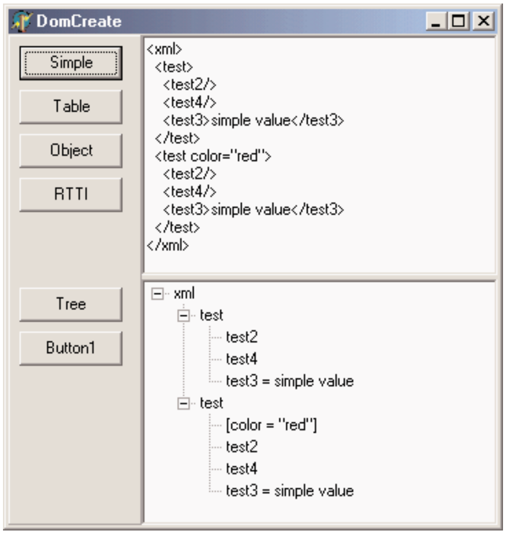
The first button of the form, Simple, creates some simple XML text using the low-level, official DOM interfaces. The program calls the `createElement` method of the document for each node, adding them as children of other nodes:

```
procedure TForm1.btnSimpleClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot, iNode, iNode2, iChild, iAttribute: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('xml'));
  // node "test"
  iNode := iRoot.appendChild (iXml.createElement ('test'));
  iNode.appendChild (iXml.createElement ('test2'));
  iChild := iNode.appendChild (iXml.createElement ('test3'));
  iChild.appendChild (iXml.createTextNode('simple value'));
  iNode.insertBefore (iXml.createElement ('test4'), iChild);

  // node replication
  iNode2 := iNode.cloneNode (True);
  iRoot.appendChild (iNode2);

  // add an attribute
  iAttribute := iXml.createAttribute ('color');
  iAttribute.nodeValue := 'red';
  iNode2.attributes.setNamedItem (iAttribute);

  // show XML in memo
  Memo1.Lines.Text := FormatXMLData (XMLDoc.XML.Text);
end;
```

Notice that text nodes are added explicitly, attributes are created with a specific create call, and that the code uses `cloneNode` to replicate an entire branch of the tree. Overall, the code is quite cumbersome to write, but after a while you might get used to this style. The effect of the program is visible (formatted in the memo and in the tree) in Figure 23.3.

FIGURE 23.3:

The DomCreate example can generate various types of XML documents using a DOM.



```
<xml>
 <test>
  <test2/>
  <test4/>
  <test3>simple value</test3>
 </test>
 <test color="red">
  <test2/>
  <test4/>
  <test3>simple value</test3>
 </test>
</xml>
```

```
⊟ xml
  ⊟ test
      test2
      test4
      test3 = simple value
  ⊟ test
      [color = "red"]
      test2
      test4
      test3 = simple value
```

The second example of DOM creation relates to a dataset. I've added to the form a BDE Table component (but any other dataset would have done) and added to a button the call to my custom `DataSetToDOM` procedure, like this:

```
DataSetToDOM ('customers', 'customer', XMLDoc, Table1);
```

The DataSetToDOM procedure creates a root node with the text of the first parameter, then grabs each record of the dataset, defines a node with the second parameter, and adds a subnode for each field of the record, all using extremely generic code:

```
procedure DataSetToDOM (RootName, RecordName: string; XMLDoc: TXMLDocument;
  DataSet: TDataSet);
var
  iNode, iChild: IXMLNode;
  i: Integer;
begin
  DataSet.Open;
  DataSet.First;
  // root
  XMLDoc.DocumentElement := XMLDoc.CreateNode (RootName);

  // add table data
  while not DataSet.EOF do
  begin
    // add a node for each record
    iNode := XMLDoc.DocumentElement.AddChild (RecordName);
    for I := 0 to DataSet.FieldCount - 1 do
    begin
      // add an element for each field
      iChild := iNode.AddChild (DataSet.Fields[i].FieldName);
      iChild.Text := DataSet.Fields[i].AsString;
    end;
    DataSet.Next;
  end;
end;
```

The preceding code uses the simplified DOM access interfaces provided by Borland, which include an AddChild node that creates the subnode, and the direct access to the Text property for defining a child node with textual content. This apparently simple routine extracts an XML representation of your dataset, also opening up a lot of opportunities for Web publishing, as I'll discuss later in the section on XSL.

Another interesting opportunity is the generation of XML documents describing Delphi objects. The DomCreate program has a button used to describe a few selected properties of an object, again using the low-level DOM:

```
procedure AddAttr (iNode: IDOMNode; Name, Value: string);
var
  iAttr: IDOMNode;
begin
  iAttr := iNode.ownerDocument.createAttribute (name);
  iAttr.nodeValue := Value;
  iNode.attributes.setNamedItem (iAttr);
end;
```

```
procedure TForm1.btnObjectClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('Button1'));

  // a few properties as attributes (might also be nodes)
  AddAttr (iRoot, 'Name', Button1.Name);
  AddAttr (iRoot, 'Caption', Button1.Caption);
  AddAttr (iRoot, 'Font.Name', Button1.Font.Name);
  AddAttr (iRoot, 'Left', IntToStr (Button1.Left));
  AddAttr (iRoot, 'Hint', Button1.Hint);

  // show XML in memo
  Memo1.Lines := XmlDoc.XML;
end;
```

Of course, it is more interesting to have a generic technique capable of saving the properties of each Delphi component (or persistent object, to be more precise), recursing on persistent subobjects and indicating the names of referenced components. This is what I've done in the ComponentToDOM procedure, which uses the low-level RTTI information provided by the TypInfo unit, including the extraction of the list of the properties of a component. Once more, the program uses the simplified Delphi XML interfaces:

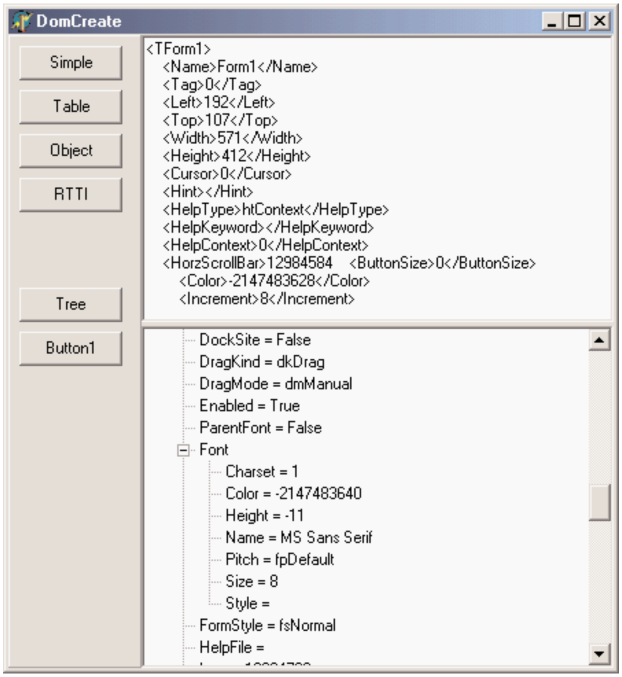
```
procedure ComponentToDOM (iNode: IXmlNode; Comp: TPersistent);
var
  nProps, i: Integer;
  PropList: PPropList;
  Value: Variant;
  newNode: IXmlNode;
begin
  // get list of properties
  nProps := GetTypeData (Comp.ClassInfo)^.PropCount;
  GetMem (PropList, nProps * SizeOf(Pointer));
  try
    GetPropInfos (Comp.ClassInfo, PropList);
    for i := 0 to nProps - 1 do
    begin
      Value := GetPropValue (Comp, PropList [i].Name);
      NewNode := iNode.AddChild(PropList [i].Name);
      NewNode.Text := Value;
      if (PropList [i].PropType^.Kind = tkClass) and (Value <> 0) then
        if TObject (Integer(Value)) is TComponent then
          NewNode.Text := TComponent (Integer(Value)).Name
        else
          // TPersistent but not TComponent: recurse
          ComponentToDOM (newNode, TObject (Integer(Value)) as TPersistent);
    end;
  finally
    FreeMem (PropList);
  end;
end;
```

These two lines of code, in this case, trigger the creation of the XML document (visible in Figure 23.4):

```
XMLDoc.DocumentElement := XMLDoc.CreateNode(self.ClassName);
ComponentToDOM (XMLDoc.DocumentElement, self);
```

FIGURE 23.4:

The XML generated to describe the form of the DomCreate program. Notice (in the tree and in the memo text) that properties of class types are further expanded.



```
<TForm1>
  <Name>Form1</Name>
  <Tag>0</Tag>
  <Left>192</Left>
  <Top>107</Top>
  <Width>571</Width>
  <Height>412</Height>
  <Cursor>0</Cursor>
  <Hint></Hint>
  <HelpType>htContext</HelpType>
  <HelpKeyword></HelpKeyword>
  <HelpContext>0</HelpContext>
  <HorzScrollBar>12984584    <ButtonSize>0</ButtonSize>
    <Color>-2147483628</Color>
    <Increment>8</Increment>
```

```
DockSite = False
DragKind = dkDrag
DragMode = dmManual
Enabled = True
ParentFont = False
Font
  Charset = 1
  Color = -2147483640
  Height = -11
  Name = MS Sans Serif
  Pitch = fpDefault
  Size = 8
  Style =
FormStyle = fsNormal
HelpFile =
```

XML Data Binding Interfaces

We have seen that working with the DOM to access or generate a document is rather tedious, because you must use positional information and not logical access to the data. Also, handling series of repeated nodes of different possible types (as in the XML sample of Listing 23.1, describing books) is far from simple. Moreover, using a DOM, you can create any well-formed document, but (unless you use a validating DOM) you can add any subnode to any node, coming up with almost useless documents, as no one else will be able to manage them.

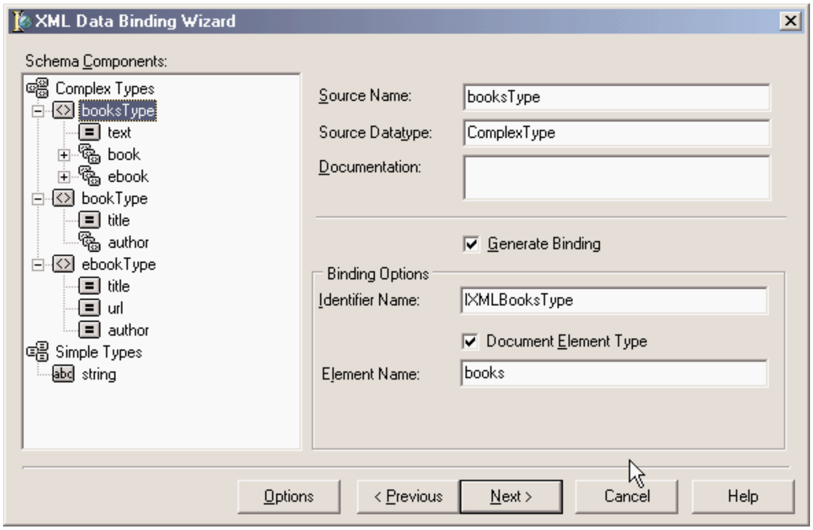
To solve these issues, Borland has added to Delphi 6 an XML Data Binding Wizard, which can examine an XML document or a document definition (a schema, a DTD, or another type of definition) and generate a set of interfaces for manipulating the document. These interfaces are specific to the document and its structure, and allow you to have more readable code, but are certainly less generic as far as the types of documents you can handle with them (and this is far more positive than it might sound at first).

You can activate the XML Data Binding Wizard by using the corresponding icon in the first page of the New Items dialog box of the IDE or by double-clicking directly on the XMLDocument component. (What is quite odd is that the corresponding command is not in the local menu of the component.)

After a first page where you can select a input file, this wizard shows you the structure of the document graphically, as you can see in Figure 23.5 for the sample XML file from Listing 23.1. In this page, you can give a name to each entity of the generated interfaces, in case you don't like the defaults suggested by the wizard. You can actually also change the rules used by the wizard to generate the names, an extended flexibility I'd like to have in other areas of the Delphi IDE. The final page gives you a preview of the generated interfaces and offers options for generating schemas and other definition files.

FIGURE 23.5:

Delphi's XML Data Binding Wizard can examine the structure of a document or a schema (or another document definition) to create a set of interfaces for simplified and direct access to the DOM data.



Source Name: booksType

Source Datatype: ComplexType

Documentation:

☑ Generate Binding

Binding Options

Identifier Name: IXMLBooksType

☑ Document Element Type

Element Name: books

For the sample XML file with the author names, the XML Data Binding Wizard generates an interface of the root node, and four interfaces for the two lists of different elements and the actual elements (books and e-books). These are a few excerpts of the generated code, available in the XmlIntfDefinition unit of the XmlInterface example:

```
type
  IXMLBooksType = interface(IXMLNode)
    ['{C9A9FB63-47ED-4F27-8ABA-E71F30BA7F11}']
    { Property Accessors }
    function Get_Text: WideString;
    function Get_Book: IXMLBookTypeList;
    function Get_Ebook: IXMLEbookTypeList;
    procedure Set_Text(Value: WideString);
    { Methods & Properties }
    property Text: WideString read Get_Text write Set_Text;
    property Book: IXMLBookTypeList read Get_Book;
    property Ebook: IXMLEbookTypeList read Get_Ebook;
  end;

  IXMLBookTypeList = interface(IXMLNodeCollection)
    ['{3449E8C4-3222-47B8-B2B2-38EE504790B6}']
    { Methods & Properties }
    function Add: IXMLBookType;
    function Insert(const Index: Integer): IXMLBookType;
    function Get_Item(Index: Integer): IXMLBookType;
    property Items[Index: Integer]: IXMLBookType read Get_Item; default;
  end;

  IXMLBookType = interface(IXMLNode)
    ['{26BF5C51-9247-4D1A-8584-24AE68969935}']
    { Property Accessors }
    function Get_Title: WideString;
    function Get_Author: IXMLString_List;
    procedure Set_Title(Value: WideString);
    { Methods & Properties }
    property Title: WideString read Get_Title write Set_Title;
    property Author: IXMLString_List read Get_Author;
  end;
```

For each interface, the XML Data Binding Wizard also generates an implementation class that provides the code for the interface methods by translating the requests into DOM calls. The unit includes three initialization functions, which can return the interface of the root node from a document loaded in an XMLDocument component (or a component providing a generic IXMLDocument interface), or return one from a file, or create a brand new DOM:

```
function Getbooks(Doc: IXMLDocument): IXMLBooksType;
function Loadbooks(const FileName: WideString): IXMLBooksType;
function Newbooks: IXMLBooksType;
```

After generating these interfaces using the wizard, in the XmlInterface example, I've repeated XML document access code similar to the XmlDomTree example, but much simpler to write (and to read). For example, you can get the attribute of the root node by writing:

```
procedure TForm1.btnAttrClick(Sender: TObject);
var
  Books: IXMLBooksType;
begin
  Books := Getbooks (XmlDocument1);
  ShowMessage (Books.Text);
end;
```

Simple, isn't it? It is even simpler if you recall that while typing this code, Delphi's code insight can help by listing the available properties of each node, thanks to the fact that the parser can read in the interface definitions (while it cannot understand the format of a generic XML document). Accessing a node of one of the sublists is a matter of writing one of the following statements (possibly the second with the default array property):

```
Books.Book.Items[1].Title  // full
Books.Book[1].Title        // further simplified
```

Similarly simplified code can be used to generate new documents or add new elements, also thanks to the customized Add method is available in each list-based interface. Again, if you don't have a predefined structure for the XML document, as in the dataset-based and RTTI-based examples of the previous demonstration, you won't be able to use this approach.

Validation and Schemas in Short

The XML Data Binding Wizard can work from existing schemas or generate one for an XML document (and eventually save it in a file with the .XDB extension). But what is a schema for, and what does it look like? An XML document describes some data, but to exchange this data among companies it has to stick to some agreed structure. A schema is a document definition, against which a document can be checked for correctness, an operation usually indicated with the term *validation*.

The first—and still very widespread—type of validation available for XML was *document type definitions (DTDs)*. These documents describe the structure of the XML but cannot really define the possible content of each node. Also, DTDs are not XML document themselves but use a different, very awkward notation.

At the end of year 2000, the W3C approved the first official draft of XML *schemas* (already available in an incompatible version called XML-Data within Microsoft's DOM). An XML schema is itself a XML document, one that can validate both the structure of the XML tree and the content of the node. A schema is based on the use and definition of simple and complex data types, similar to what happens in an OOP programming language.

A schema defines complex types, indicating for each the possible nodes, their optional sequence (sequence, all), the number of occurrences for each subnode (minOccurs, maxOccurs), and the data type of each specific element. Here is the schema defined by the XML Data Binding Wizard for the usual sample books file:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://www.borland.com/schemas/delphi/6.0/XMLDataBinding">
  <xs:element name="books" type="booksType"/>
  <xs:complexType name="booksType">
    <xs:annotation>
      <xs:appinfo xdb:docElement="books"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="book" type="bookType" maxOccurs="unbounded"/>
      <xs:element name="ebook" type="ebookType" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="text" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="bookType">
    <xs:annotation>
      <xs:appinfo xdb:repeated="True"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ebookType">
    <xs:annotation>
      <xs:appinfo xdb:repeated="True"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="url" type="xs:string"/>
      <xs:element name="author" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

NOTE As I write, there are still very few DOM implementations that can be used to validate a document against an XML schema. Apache Xerces DOM has good support for schemas. Another tool I've used for validation is XSV (XML Schema Validator), an open source attempt at a conformant schema-aware processor, which can be used either directly via the Web or after downloading a command-line executable.

Using the SAX API

The Simple API for XML, or SAX, doesn't create a tree for the XML nodes, but simply parses the node-firing events for each node, attribute, value, and so on. Because it doesn't keep the document in memory, using the SAX allows managing much larger documents. Its approach is also very useful for one-time examination of a document, or retrieval of specific information. This is a list of events fired by the SAX:

- `StartDocument` and `EndDocument` for the entire document
- `StartElement` and `EndElement` for each node
- `Characters` for the text within the nodes

It is quite common to use a stack to handle the current path within the nodes tree, and push and pop elements to and from the stack for every `StartElement` and `EndElement` event.

Delphi 6 does not include specific support for the SAX interface, but this can be easily obtained by importing Microsoft's XML support (the MSXML library). In particular for the SaxDemo1 example I've used version 2 of MSXML: the Pascal version of its type library is available within the source code of the program, but you must have the COM library registered on your computer to run the program successfully.

To use the SAX, you have to install a SAX event handler within a SAX reader, then load a file and parse it. I've used the SAX reader interface provided by MSXML for VB programmers. In fact, the official (C++) interface had a few errors in its type library that prevented Delphi from importing it properly (the newer MSXML 3 might have fixed this issue by the time you read this).

In the main form of the SaxDemo1 example, I've declared:

```
sax: IVBSAXXMLReader;
```

In the `FormCreate` method, this is initialized with the actual COM object:

```
sax := CreateComObject (CLASS_SAXXMLReader) as IVBSAXXMLReader;
sax.ErrorHandler := TMySaxErrorHandler.Create;
```

The code also sets an error handler, which is a class implementing a specific interface, `IVBSAXErrorHandler`, with three methods that are called depending on the severity of the problem: `error`, `fatalError`, and `ignorableWarning`.

Simplifying the code a little, the SAX parser is activated by calling the `parseURL` method after assigning a content handler to it:

```
sax.ContentHandler := TMySaxHandler.Create;
sax.parseURL (filename)
```

So the code ultimately resides in the `TMySaxHandler` class, which has the SAX events. Because I have multiple SAX content handlers in this example, I've written a base class with the core code and a few specialized versions for specific processing; this is the code of the base class, which implements both the `IDispatch` and `IVBSAXContentHandler` interfaces:

```
type
  TMySaxHandler = class (TInterfacedObject, IVBSAXContentHandler)
  protected
    stack: TStringList;
  public
    constructor Create;
    destructor Destroy; override;
    // IDispatch
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
      HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer):
      HResult; stdcall;
    // IVBSAXContentHandler
    procedure Set_documentLocator(const Param1: IVBSAXLocator);
      virtual; safecall;
    procedure startDocument; virtual; safecall;
    procedure endDocument; virtual; safecall;
    procedure startPrefixMapping(var strPrefix: WideString;
      var strURI: WideString); virtual; safecall;
    procedure endPrefixMapping(var strPrefix: WideString); virtual; safecall;
    procedure startElement(var strNamespaceURI: WideString;
      var strLocalName: WideString; var strQName: WideString;
      const oAttributes: IVBSAXAttributes); virtual; safecall;
    procedure endElement(var strNamespaceURI: WideString;
      var strLocalName: WideString; var strQName: WideString);
      virtual; safecall;
    procedure characters(var strChars: WideString); virtual; safecall;
    procedure ignorableWhitespace(var strChars: WideString);
      virtual; safecall;
    procedure processingInstruction(var strTarget: WideString;
      var strData: WideString); virtual; safecall;
    procedure skippedEntity(var strName: WideString); virtual; safecall;
  end;
```

The most interesting portion, of course, is the final list of SAX events. All this base class does is emit information to a log when the parser starts (`startDocument`) and finishes (`endDocument`) and keep track of the current node and all of its parent nodes with a stack:

```
// TMySaxHandler.startElement
stack.Add (strLocalName);
// TMySaxHandler.endElement
```

```
stack.Delete (stack.Count - 1);
```

An actual implementation is provided by the `TMySimpleSaxHandler` class, which overrides the `startElement` event triggered for any new node to output the current position in the tree with the statement:

```
Log.Add (strLocalName + '(' + stack.CommaText + ')');
```

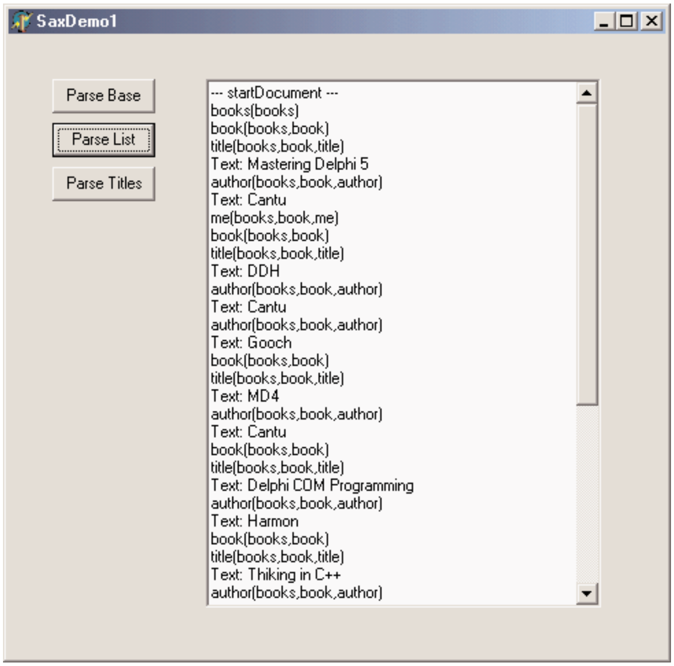
The second method of the class is the `characters` event, triggered when a node value (or a test node) is encountered, to output its content:

```
procedure TMySimpleSaxHandler.characters(var strChars: WideString);
var
  str: WideString;
begin
  inherited;
  str := RemoveWhites (strChars);
  if (str <> '') then
    Log.Add ('Text: ' + str);
end;
```

The two methods produce the combined effect of Figure 23.6.

FIGURE 23.6:

The log produced by reading an XML document with the SAX in the SaxDemo1 example



```
--- startDocument ---
books(books)
book(books,book)
title(books,book,title)
Text: Mastering Delphi 5
author(books,book,author)
Text: Cantu
me(books,book,me)
book(books,book)
title(books,book,title)
Text: DDH
author(books,book,author)
Text: Cantu
author(books,book,author)
Text: Gooch
book(books,book)
title(books,book,title)
Text: MD4
author(books,book,author)
Text: Cantu
book(books,book)
title(books,book,title)
Text: Delphi COM Programming
author(books,book,author)
Text: Harmon
book(books,book)
title(books,book,title)
Text: Thiking in C++
author(books,book,author)
```

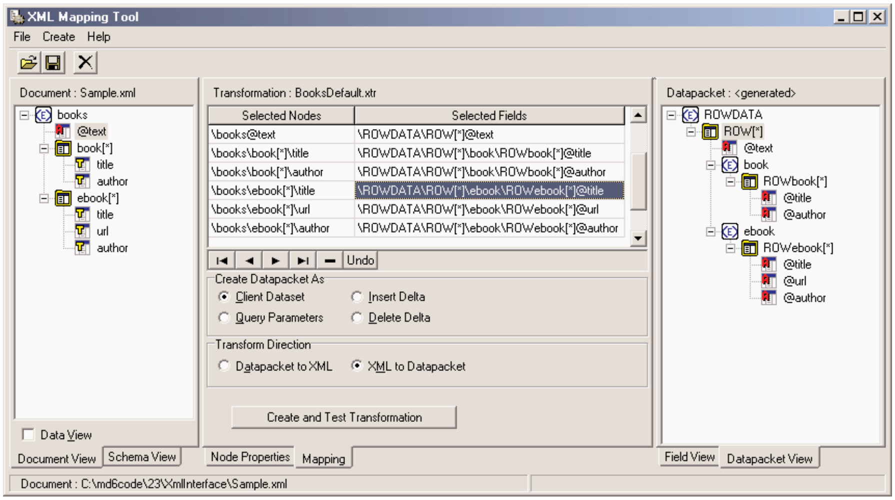
This is still a generic parsing operation affecting the entire XML file. The second derived SAX content handler class, instead, refers to the specific structure of the XML document, extracting only nodes of a give type. In particular, the program looks for nodes of the *title* type. When a node has this type (in `startElement`), the class sets the `isbook` Boolean variable. The text value of the node is considered only right after a node of this type is encountered:

```
procedure TMyBooksListSaxHandler.startElement(var strNamespaceURI,
  strLocalName, strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
  inherited;
  isbook := (strLocalName = 'title');
end;

procedure TMyBooksListSaxHandler.characters(var strChars: WideString);
var
  str: string;
begin
  inherited;
  if isbook then
  begin
    str := RemoveWhites (strChars);
    if (str <> '') then
      Log.Add (stack.CommaText + ': ' + str);
  end;
end;
```

Mapping XML with Transformations

There is one more technique you can use in Delphi 6 to handle at least some XML documents. You can create a *transformation* to translate the XML of a generic document into the format used natively by the ClientDataSet component when saving data to a MyBase XML file. In the reverse direction, another transformation can turn a dataset available within a ClientDataSet (through a DataSetProvider component) into a XML file of a required format (or schema).

Delphi 6 includes a wizard to generate such transformations. Called XML Mapping Tool, or XML Mapper for short, it is invokable from the Tools menu of the IDE or executed as a stand-alone application. The XML Mapper, visible in Figure 23.7, is a design-time helper that assists you in defining transformation rules between the nodes of a generic XML document and fields of the data packet of the ClientDataSet.



Selected Nodes	Selected Fields
\books@text	\ROWDATA\ROW[*]@text
\books\book[*]\title	\ROWDATA\ROW[*]\book\ROWbook[*]@title
\books\book[*]\author	\ROWDATA\ROW[*]\book\ROWbook[*]@author
\books\ebook[*]\title	\ROWDATA\ROW[*]\ebook\ROWebook[*]@title
\books\ebook[*]\url	\ROWDATA\ROW[*]\ebook\ROWebook[*]@url
\books\ebook[*]\author	\ROWDATA\ROW[*]\ebook\ROWebook[*]@author

Create Datapacket As

● Client Dataset ○ Insert Delta

○ Query Parameters ○ Delete Delta

Transform Direction

○ Datapacket to XML ● XML to Datapacket

Document : C:\md6code\23\XmlInterface\Sample.xml

FIGURE 23.7:

The XML Mapper shows the two sides of a transformation to define a mapping among them (with the rules indicated in the central portion).

The XML Mapper windows has three areas:

- On the left is the XML document section, which displays information about the structure of the XML document (and eventually its data, if the related check box is active) in the Document View or an XML schema in the Schema View, depending on the selected tab.
- On the right is the data packet section, which displays information about the metadata in the data packet, either in the Field View indicating the dataset structure or in the Datapacket View reporting the XML structure. Notice, in fact, that the XML Mapper can also open files in the native ClientDataSet format.
- The central portion of the windows is used by the mapping section. This contains two pages as well: one for Mapping, where you can see the correspondences between selected elements of the two sides that will be part of the mapping, and one for Node Properties, where you can modify the data types and other details of each of the possible mappings.

The Mapping page of the central pane also hosts the local menu used to generate the transformation, while each other pane and view has specific local menus you can use to perform the various actions (beside the few commands in the main menu).

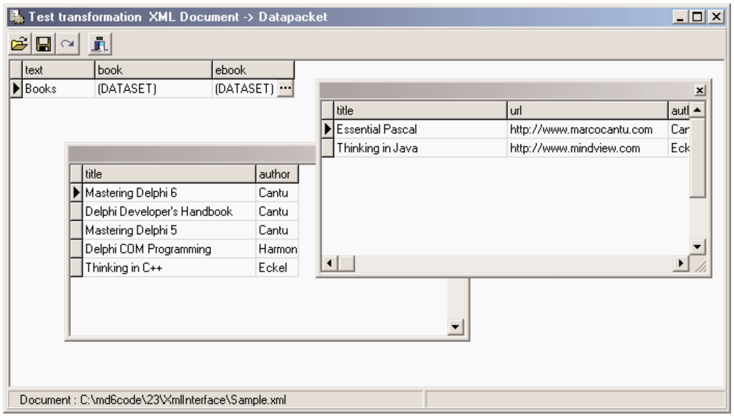
You can use XML Mapper to map an existing schema (or extract it from a document) to a brand new data packet, an existing data packet to a new schema or document, or an existing data packet into an existing XML document (if a match is reasonable). Besides converting the data of an XML file into a data packet, you can also convert to a delta packet of the ClientDataSet. This is useful for merging a document to an existing table, as if a user had inserted them. In particular, you can transform an XML document into a delta packet for records to be modified, deleted, or inserted.

The result of using the XML Mapper is one or more transformation files, each representing a one-way conversion (so you need at least two transformation files to convert data back and forth). These transformation files are then used at design time and at run time by the XMLTransform, XMLTransformProvider, and XMLTransformClient components.

As an example, I've tried opening the usual "books" XML, which has a structure that doesn't easily match a table, since there are two lists of values of different types (I've skipped easier examples in which the XML has a plain rectangular structure). After opening the `Sample.XML` file in the XML Document section, I've used its local menu to select all of its elements (Select All) and to create the data packet (Create Datapacket From XML). This automatically fills the right pane with the data packet and the central portion with the proposed transformation. You can also immediately view its effect in a sample program by selecting the Create And Test Transformation button. This opens a generic application that can load a document into the dataset using the transformation you've just created, as you can see in Figure 23.8.

FIGURE 23.8:

The Create and Test Transformation button of the XML Mapper tool allows you to immediately verify the effect of the transformations you are building



Test transformation XML Document -> Datapacket

text	book	ebook
Books	(DATASET)	(DATASET)

title	author
Mastering Delphi 6	Cantu
Delphi Developer's Handbook	Cantu
Mastering Delphi 5	Cantu
Delphi COM Programming	Harmon
Thinking in C++	Eckel

title	url	autl
Essential Pascal	http://www.marcocantu.com	Car
Thinking in Java	http://www.mindview.com	Eck

Document : C:\md6code\23\XmlInterface\Sample.xml

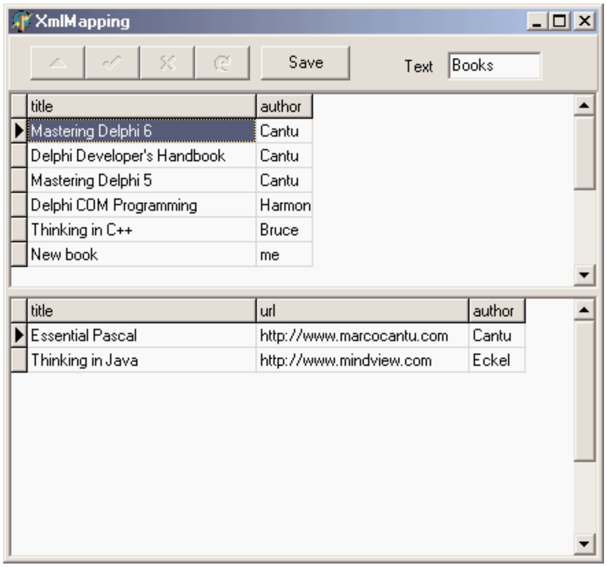
In this specific case, you can see that the XML Mapper generates a table with two dataset fields, one of each possible list of subelements. This was the only possible standard solution, as the two sublists have different structures, and is the only solution that allows you to edit the data in a DBGrid attached to the ClientDataSet and save it back to a complete XML file, as demonstrated by the XmlMapping example. This program is basically a Windows-based editor of a complex XML document.

It uses a TransformProvider component, with two transformation files attached, to read in an XML document and make it available to a ClientDataSet. As the name suggests, in fact, this component is a dataset provider. To build the user interface, I haven't connected the

ClientDataSet directly to a grid, as it has a single record with a text field plus two detailed datasets. For this reason, I've added to the program two more ClientDataSet components, attached to the dataset fields and connected to two DBGrid controls. This is probably easier to understand by looking at its DFM source code in the following excerpt and at its output in Figure 23.9.

FIGURE 23.9:

The XmlMapping example uses a TransformProvider component to make a complex XML document available for editing within multiple ClientDataSet components.



Text: Books

title	author
Mastering Delphi 6	Cantu
Delphi Developer's Handbook	Cantu
Mastering Delphi 5	Cantu
Delphi COM Programming	Harmon
Thinking in C++	Bruce
New book	me

title	url	author
Essential Pascal	http://www.marcocantu.com	Cantu
Thinking in Java	http://www.mindview.com	Eckel

```
object Form1: TForm1
  Caption = 'XmlMapping'
  object XMLTransformProvider1: TXMLTransformProvider
    TransformRead.TransformationFile = 'BooksDefault.xtr'
    TransformWrite.TransformationFile = 'BooksDefaultToXml.xtr'
    XMLDataFile = 'Sample.xml'
  end
  object ClientDataSet1: TClientDataSet
    ProviderName = 'XMLTransformProvider1'
    object ClientDataSet1text: TStringField
      FieldName = 'text'
      Size = 5
    end
    object ClientDataSet1book: TDataSetField
      FieldName = 'book'
    end
    object ClientDataSet1ebook: TDataSetField
      FieldName = 'ebook'
    end
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
  object Panel1: TPanel
    Align = alTop
    object Label2: TLabel
      Caption = 'Text'
      FocusControl = DBEdit2
    end
    object DBNavigator1: TDBNavigator
      VisibleButtons = [nbEdit, nbPost, nbCancel, nbRefresh]
    end
    object DBEdit2: TDBEdit
      DataField = 'text'
      DataSource = DataSource1
    end
    object Button1: TButton
      Caption = 'Save'
      OnClick = Button1Click
    end
  end
  object ClientDataSet2: TClientDataSet
    DataSetField = ClientDataSet1book
  end
  object DataSource2: TDataSource
    DataSet = ClientDataSet2
  end
  object DBGrid1: TDBGrid
    Align = alTop
    DataSource = DataSource2
  end
  object Splitter1: TSplitter
    Cursor = crVSplit
    Align = alTop
  end
  object ClientDataSet3: TClientDataSet
    DataSetField = ClientDataSet1ebook
  end
  object DataSource3: TDataSource
    DataSet = ClientDataSet3
    Left = 232
    Top = 224
  end
  object DBGrid2: TDBGrid
    Align = alClient
    DataSource = DataSource3
  end
end
```

This program allows you to edit the data of the various sublists of nodes, within the grids, modifying them but also adding or deleting records. As you apply the changes to the dataset (clicking the Save button, which calls `ApplyUdpates`), the transform provider saves an updated version of the file to disk.

As an alternative approach, you can also create transformations that map only portions of the XML document into a dataset. As an example, see the `BooksOnly.xtr` file in the folder of the XmlMapping example. This can be useful for viewing the data, but the modified XML document you'll generate will have a different structure and content from the original, including only the portion you've selected. So this can be useful for viewing the data, but not for editing it.

NOTE It is not surprising that the transformation files are themselves XML documents, as you can see by opening one in the editor. This XML document uses a custom format.

At the opposite side, we can see how a transformation can be used to take a database table or the result of a query and produce an XML file with a more readable format than the one provided by default by the ClientDataSet persistence mechanism. To build the MapTable example, I've placed a table component on a form and attached a DataSetProvider to it and a ClientDataSet to the provider. After opening the table and the client dataset, I saved its content to an XML file.

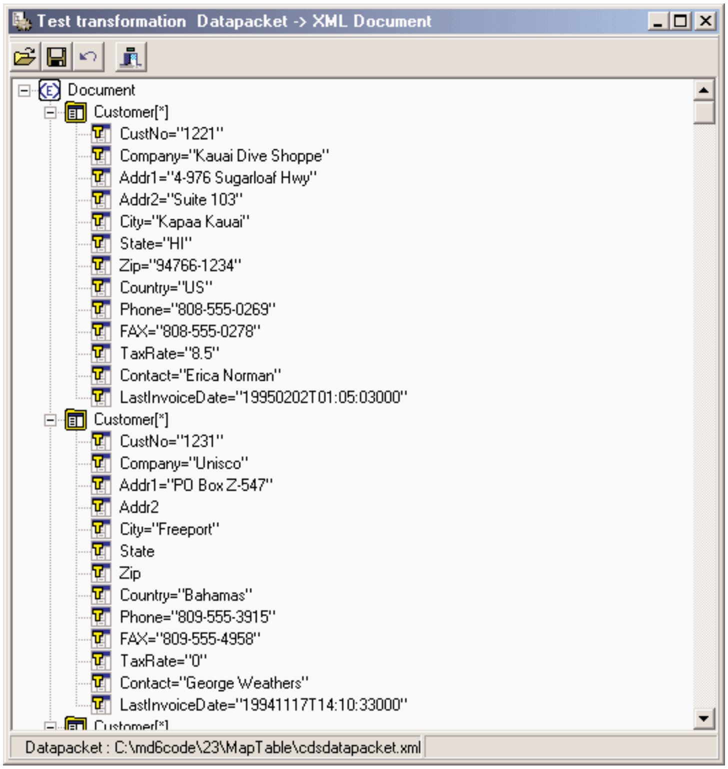
At that point, I opened the XML Mapper, loaded the data packet file into it, selected all of the data packet nodes (with the Select All command of its local menu) and invoked the Create XML From Datapacket command. In the following dialog box, I accepted the default name mappings for fields and only changed the suggested name for record nodes (*ROW*) into something more readable (*Customer*). If you now test the transformation, the XML Mapper will display the contents of the resulting XML document in a custom tree view, as you can see in Figure 23.10.

After saving the transformation file, I was ready to resume developing the program, adding to it another provider that takes the data from the ClientDataSet (as a user might edit in on an attached DBGrid before transforming it) and makes it available to an XMLTransform-Client component. This component has the transformation file connected to it, but not an XML file. In fact, clicking the button shows the XML document within a memo (after a formatting it) instead of saving it to a file, something you can do by calling the `GetDataAsXml` method (even if the Help file is far from clear about this):

```
procedure TForm1.btnMapClick(Sender: TObject);
begin
  Memo1.Lines.Text := FormatXmlData(XMLTransformClient1.GetDataAsXml(''));
end;
```

FIGURE 23.10:

When you convert from a data packet to an XML document, the XML Mapper can preview the document in a tree structure.

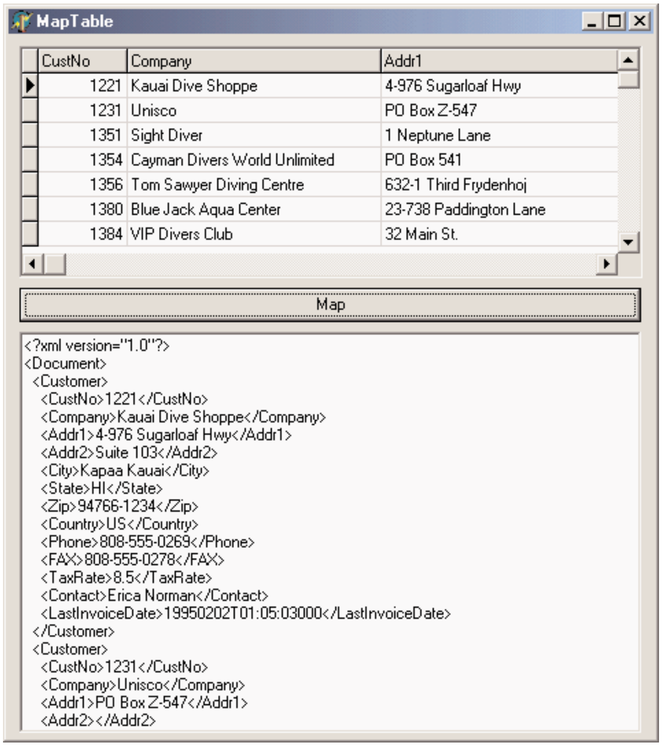


Test transformation Datapacket -> XML Document

```
Document
  Customer[*]
    CustNo="1221"
    Company="Kauai Dive Shoppe"
    Addr1="4-976 Sugarloaf Hwy"
    Addr2="Suite 103"
    City="Kapaa Kauai"
    State="HI"
    Zip="94766-1234"
    Country="US"
    Phone="808-555-0269"
    FAX="808-555-0278"
    TaxRate="8.5"
    Contact="Erica Norman"
    LastInvoiceDate="19950202T01:05:03000"
  Customer[*]
    CustNo="1231"
    Company="Unisco"
    Addr1="PO Box Z-547"
    Addr2
    City="Freeport"
    State
    Zip
    Country="Bahamas"
    Phone="809-555-3915"
    FAX="809-555-4958"
    TaxRate="0"
    Contact="George Weathers"
    LastInvoiceDate="19941117T14:10:33000"
  Customer[*]
```

Datapacket : C:\md6code\23\MapTable\cdsdatapacket.xml

This is the only code of the program visible at run time in Figure 23.11. The application has much simpler code than the DomCreate example I used to generate a similar XML document, but requires the design-time definition of the transformation. The DomCreate example, instead, could work on any dataset at run time, without any connection to a specific table, as it had some rather generic code. In theory, it is possible to produce similar dynamic mappings by using the events of the generic XMLTransform component, but I find it much easier to use the DOM-based approach discussed earlier. Notice also that the `FormatXmlData` call produces much nicer output but slows down the program, because it involves loading the XML into a DOM.



CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj
1380	Blue Jack Aqua Center	23-738 Paddington Lane
1384	VIP Divers Club	32 Main St.

```
<?xml version="1.0"?>
<Document>
 <Customer>
  <CustNo>1221</CustNo>
  <Company>Kauai Dive Shoppe</Company>
  <Addr1>4-976 Sugarloaf Hwy</Addr1>
  <Addr2>Suite 103</Addr2>
  <City>Kapaa Kauai</City>
  <State>HI</State>
  <Zip>94766-1234</Zip>
  <Country>US</Country>
  <Phone>808-555-0269</Phone>
  <FAX>808-555-0278</FAX>
  <TaxRate>8.5</TaxRate>
  <Contact>Erica Norman</Contact>
  <LastInvoiceDate>19950202T01:05:03000</LastInvoiceDate>
 </Customer>
 <Customer>
  <CustNo>1231</CustNo>
  <Company>Unisco</Company>
  <Addr1>PO Box Z-547</Addr1>
  <Addr2></Addr2>
```

FIGURE 23.11:

The MapTable example can generate an XML document from a database table using a custom transformation file. You can see the original dataset in the DBGrid above and the resulting XML document in the memo control below

XML and Internet Express

Once you have defined the structure of an XML document, you might want to let users see and edit the data in a Windows application or over the Web. This second case is rather interesting, as Delphi provides specific support for it. Delphi 5 already included an architecture called Internet Express, which is now available in Delphi 6 as part of the WebSnap platform. WebSnap offers also support for XSL, which I'll discuss later.

In Chapter 17, "Multitier Database Applications with DataSnap," I discussed the development of DataSnap applications (formerly known as Midas applications). Internet Express provides a client component for this architecture, called XMLBroker, which can be used in place of a client dataset to retrieve data from a middle-tier DataSnap program and make it available to a specific type of page producer, called InetXpageProducer. You can use these components in a standard WebBroker application or in a WebSnap program. The idea behind Internet Express is that you write a Web server extension (CGI or ISAPI or Apache modules, as discussed in the preceding chapter), which in turn produces Web pages hooked to your DataSnap server. Your custom application acts as a DataSnap client and produces pages for a browser client. Internet Express offers the services required to build this custom application easily.

I know this sounds confusing, but Internet Express is a four-tier architecture: SQL server, application server (the DataSnap server), Web server with a custom application, and finally Web browser. Of course, you can place the database access components within the same application handling the HTTP request and generating the resulting HTML, as in a client/server solution. You can even access a local database or an XML file, in a single-tier structure.

In other words, Internet Express is a technology for building clients based on a browser, which lets you send the entire dataset to the client computer along with the HTML and some JavaScript for manipulating the XML and showing it into the user interface defined by the HTML. It is the JavaScript that enables the browser to show the data and manipulate it.

The XMLBroker Component

Internet Express uses multiple technologies to accomplish this. The DataSnap data packets are converted into the XML format, to let the program embed this data right into the HTML page for Web client-side manipulation. Actually, the Delta data packet is also represented in XML. These operations are performed by the XMLBroker component, which can handle XML and provide data to the new JavaScript components. Like the ClientDataSet, the XMLBroker has

- A `MaxRecords` property indicating the number of records to add to a single page
- A `Params` property hosting the parameters that components will forward to the remote query through the provider
- A `WebDispatch` property indicating the update request the broker responds to

The InetXPageProducer allows you to generate HTML forms from datasets, in a visual way similar to the development of an AdapterPageProducer user interface. Actually, the Internet Express architecture, the interfaces it uses internally, and some of its IDE editor can together be considered the parent of the WebSnap architecture. With the notable difference of generating scripts to be executed on the server side and on the client side, they both provide an editor for placing visual components and generating such scripts. A notable difference I'm personally not terribly happy about, though, is that the older Internet Express is more XML-oriented than newer WebSnap.

TIP Another common feature of the InetXPageProducer and the AdapterPageProducer is the support for Cascading Style Sheets (CSS). These components have the two alternative `Style` and `StylesFile` properties for defining the CSS, and each visual element has a `StyleRule` property where you can select the style name.

JavaScript Support

To make the editing operations on the client side powerful, the InetXPageProducer uses special JavaScript components and code. Delphi embeds a rather large JavaScript library, which the browser will have to download. This might seem a nuisance, but it is the only way the browser interface (which is based on dynamic HTML) can be rich enough to support field constraints and other business rules with the browser. This is really impossible with plain HTML. The JavaScript files provided by Borland, and that you should make available on the Web site hosting the application, are the following:

File	Description
`Xmldom.js`	DOM-compatible XML parser (for browsers lacking native XML DOM support)
`Xmldb.js`	JavaScript classes for the HTML controls
`Xmldisp.js`	JavaScript classes for binding XML data with the HTML controls
`Xmlerrdisp.js`	Classes for reconciling errors
`XmlShow.js`	JavaScript functions to display data and delta packets (for debugging purposes)

HTML pages generated by Internet Express usually include references to these JavaScript files, such as:

```
<script language=Javascript type="text/javascript"
  src="IncludePathURL/xmldb.js"></script>
```

You can customize the JavaScript by adding code directly into the HTML pages or by creating a new Delphi components, written to fit with the Internet Express architecture that “emits” JavaScript code (possibly along with HTML). As an example, the sample `TPromptQueryButton` class of INetXCustom generates the following HTML and JavaScript code:

```
<script language=javascript type="text/javascript">
  function PromptSetField(input, msg) {
    var v = prompt(msg);
    if (v == null || v == "")
      return false;
    input.value = v
    return true;
  }
  var QueryForm3 = document.forms['QueryForm3'];
</script>
<input type=button value="Prompt..."
  onclick="if (PromptSetField(PromptResult, 'Enter some text\n'))
    QueryForm3.submit();">
```

TIP If you plan on using Internet Express, have a look at the INetXCustom extra demo components, available in the `\Demos\Midas\InternetExpress\INetXCustom` folder. Follow the detailed instructions in the `readme.txt` file for the installation of these components, which are provided by Borland with no support but allow you to add many more features to your Internet Express applications with little extra effort.

Of course, to deploy this architecture you don't need anything special on the client side, as any browser up to the HTML 4 standard can be used, on any operating system. The Web server, instead, must be a Win32 server (we're waiting for this technology to be available in Kylix) and you must deploy the DataSnap libraries on it (after paying the proper license fee to Borland, still not disclosed at this time).

Building a First Example

To better understand what I'm talking about, and as a way to cover some more technical details, let me try out a simple demo, called IeFirst. To avoid configuration issues, this is a CGI application accessing a dataset directly—in this case, a local table retrieved via the BDE. Later I'll show you how to turn an existing DataSnap Windows client to a browser-based interface. To build IeFirst, I've created a new CGI application and added to its data module a Table and a DataSetProvider. The next step is to add an XMLBroker component and connect it to the provider:

```
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'employee.db'
end
object DataSetProvider1: TDataSetProvider
  DataSet = Table1
end
object XMLBroker1: TXMLBroker
  ProviderName = 'DataSetProvider1'
  WebDispatch.MethodType = mtAny
  WebDispatch.PathInfo = 'XMLBroker1'
  ReconcileProducer = PageProducer1
  OnGetResponse = XMLBroker1GetResponse
end
```

The `ReconcileProducer` property is required to show a proper error message in case of an update conflict. As we'll see later, one of the Delphi demos includes some custom code, but in this simple example I've simply connected a traditional PageProducer component with a generic HTML error message. After setting up the XML broker, you can add an InetXPageProducer to the Web data module. This component has a standard HTML skeleton; I've customized to add a title to it, without touching the special tags:

```
<HTML><HEAD>
  <title>IeFirst</title>
</HEAD><BODY>
  <h1>Internet Express First Demo (IeFirst.exe)</h1>
  <#INCLUDES><#STYLES><#WARNINGS><#FORMS><#SCRIPT>
</BODY>
```

The special tags are automatically expanded using the JavaScript files of the directory specified by the `IncludePathURL` property. You *must* set this property to refer to the Web server directory where these files reside. You can find them in the `\Delphi6\Source\WebMidas` directory. The five tags have the following effect:

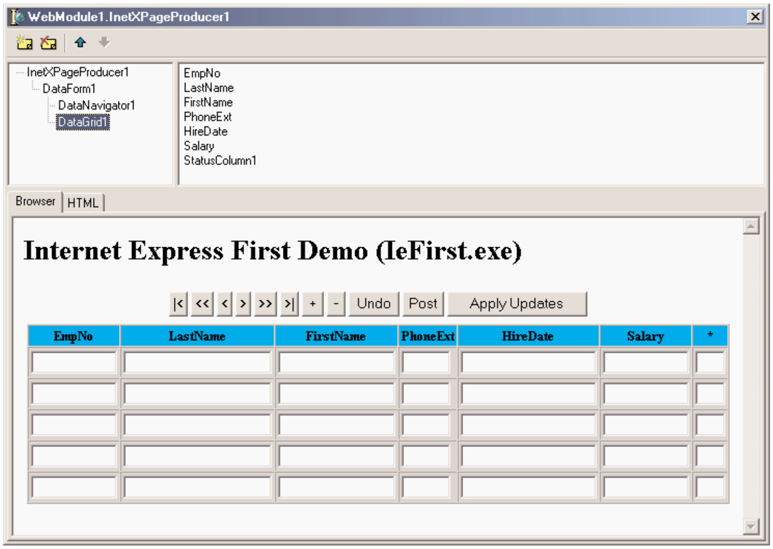
Tag	Effect
`<#INCLUDES>`	Generates the instructions to include the JavaScript libraries
`<#STYLES>`	Adds the embedded style sheet definition
`<#WARNINGS>`	Used at design time to show errors in the InetXPageProducer editor
`<#FORMS>`	Generates the HTML code produced by the components of the Web page
`<#SCRIPT>`	Adds a JavaScript block used to start up the client-side script

NOTE The InetXPageProducer component handles also a few more internal tags. `<#BODYELEMENTS>` corresponds to all of the five tags of the predefined template. `<#COMPONENT Name=WebComponentName>` is part of the generated HTML code used to declare the components generated visually. `<#DATAPACKET XMLBroker=BrokerName>` is replaced with the actual XML of the data packet.

To customize the resulting HTML of the InetXPageProducer, you can use its editor, which again is similar to the one for WebSnap server-side scripting. Just double-click the InetXPageProducer component, and Delphi opens up a window like the one in Figure 23.12 (with the final settings of the example). In this editor, you can create complex structures, starting with a query form, data form, or generic layout group. In the data form of my simple example, I've added a DataGrid and a DataNavigator component, without customizing them any further (an operation you do by adding child buttons, columns, and other objects, which fully replace the default ones).

FIGURE 23.12:

The InetXPageProducer editor allows you to build complex HTML forms visually, similarly to the AdapterPageProducer.



Internet Express First Demo (IeFirst.exe)

EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary	*

The DFM code for the InetXPageProducer and its internal components in my example is the following, where you can seen the core settings plus some limited graphical customizations:

```
object InetXPageProducer1: TInetXPageProducer
  IncludePathURL = '/jssource/'
  HTMLDoc.Strings = (...)
  object DataForm1: TDataForm
    object DataNavigator1: TDataNavigator
      XMLComponent = DataGrid1
      Custom = 'align="center"'
    end
    object DataGrid1: TDataGrid
      XMLBroker = XMLBroker1
      DisplayRows = 5
      TableAttributes.BgColor = 'Silver'
      TableAttributes.CellSpacing = 0
      TableAttributes.CellPadding = 2
      HeadingAttributes.BgColor = 'Aqua'
      object EmpNo: TTextColumn...
      object LastName: TTextColumn...
      object FirstName: TTextColumn...
      object PhoneExt: TTextColumn...
      object HireDate: TTextColumn...
      object Salary: TTextColumn...
      object StatusColumn1: TStatusColumn...
    end
  end
end
```

But the value of these components is in the HTML (and JavaScript) code they generate, which you can preview by selecting the HTML tab of the InetXPageProducer editor. Here are a few portions of the definitions in the HTML, for the buttons, the data grid heading, and one if its cells:

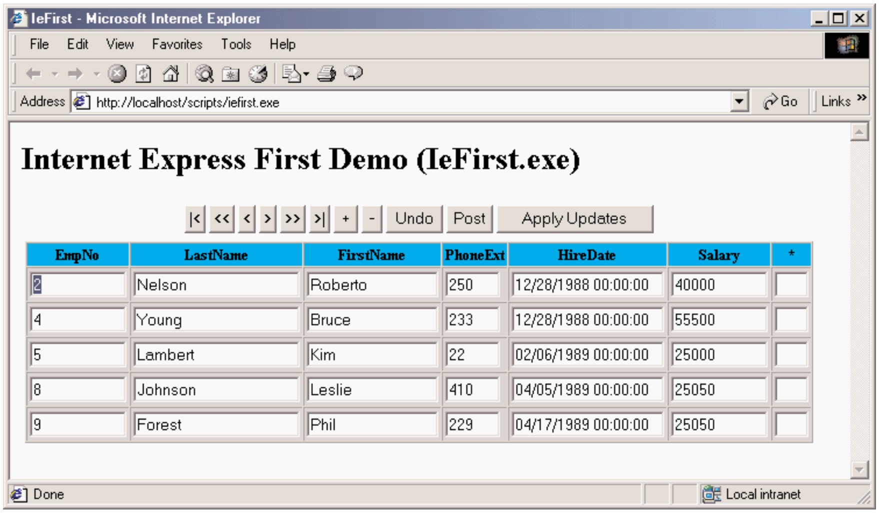
```
// buttons
<table align="center">
  <tr><td colspan="2">
    <input type="button" value="|<"
      onclick='if (xml_ready) DataGrid1_Disp.first();'>
    <input type="button" value="<<"
      onclick='if (xml_ready) DataGrid1_Disp.pgup();'>
...
// data grid heading
<table cellspacing="0" cellpadding="2" border="1" bgcolor="silver">
  <tr bgcolor="aqua">
    <th>EmpNo</th>
    <th>LastName</th>
  ...
  </tr>
  <tr>
    // a data cell
    <td><div>
      <input type="text" name="EmpNo" size="10"
        onfocus='if(xml_ready)DataGrid1_Disp.xfocus(this);'
        onkeydown='if(xml_ready)DataGrid1_Disp.keys(this);'>
    </div></td>...
```

When the HTML generator is set up, you can go back to the Web data module, add an action to it, and connect the action with the InetXPageProducer via the `Producer` property. This should be enough to make the program work through a browser, as you can see in Figure 23.13.

If you look at the HTML file received by the browser, you'll find the table mentioned in the preceding definition, some JavaScript code here and there, and the database data in the data packet XML format. This data is assembled by the XML broker and passed to the producer component to be embedded in the HTML file. Notice that the number of records sent to the client depends on the XMLBroker, not on the number of lines in the grid. Once the XML data is sent to the browser, in fact, you can use the buttons of the navigator component to move around in the data without requiring further access to the server to fetch more. This is quite different from the paging behavior of WebSnap. Not that one is better than the other; this depends on the specific application you are building.

FIGURE 23.13:

The IeFirst application sends to the browser some HTML components, an entire XML document, and the JavaScript code to show the data in the visual components.


IeFirst - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/scripts/iefirst.exe
Go
Links
|< << < > >> >| + - Undo Post Apply Updates
Done
Local intranet

Internet Express First Demo (IeFirst.exe)

EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary	*
2	Nelson	Roberto	250	12/28/1988 00:00:00	40000	
4	Young	Bruce	233	12/28/1988 00:00:00	55500	
5	Lambert	Kim	22	02/06/1989 00:00:00	25000	
8	Johnson	Leslie	410	04/05/1989 00:00:00	25050	
9	Forest	Phil	229	04/17/1989 00:00:00	25050	

At the same time, the JavaScript classes in the system allow the user to type in new data, following the rules imposed by the JavaScript code hooked to dynamic HTML events. Notice that the grid, by default, has an extra asterisk column, indicating which records have been modified. The update data is collected in an XML data packet in the browser, and sent back to the server when the user clicks the Apply Updates button. At this point, the browser activates the action specified by the `WebDispath.PathInfo` property of the XMLBroker. There is no need to export this action from the Web data module, as this operation is automatic (although you can disable it by setting `WebDispath.Enable` to False).

The XMLBroker applies the changes to the server, returning the content of the provider connected to the `ReconcileProvider` property (or raising an exception if this is not defined). When everything works fine, the XMLBroker redirects the control to the main page that contains the data. However, I've experienced some problems with this technique, so the IeFirst example handles the `OnGetReponse`, indicating this is an update view:

```
procedure TWebModule1.XMLBroker1GetResponse(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<h1>Updated</h1><p>' + InetXPageProducer1.Content;
  Handled := True;
end;
```

Master/Detail in Internet Express

My second Internet Express example goes a little beyond the basics by providing a master/detail data packet for Web browsing obtained through a DataSnap connection. The program uses the AppPlus server of Chapter 17, which defines the master/detail relationship between two tables. The dataset field embedded in the table will be transformed into a nested XML structure, delivering the same information.

The program uses a combination of XMLBroker, InetXPageProducer, and DCOMConnection. This time, I've customized the Web components, by adding a LayoutGroup component to obtain multiple columns, and I've also created fields to select the information to display. In Listing 23.2 are some snippets of the DFM file of this Web module: I've removed a lot of extra information, but I think it is worth looking at it.

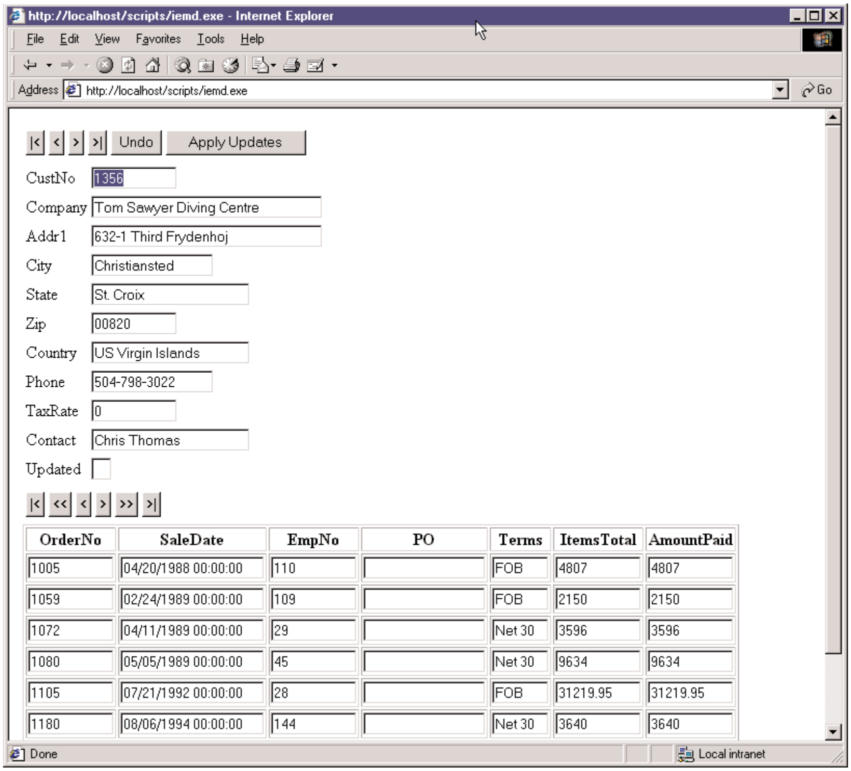
Listing 23.2: Portions of the DFM file for the IeMd example's Web module

```
object DCOMConnection1: TDCOMConnection
  Connected = False
  ServerName = 'AppSPlus.AppServerPlus'
end
object XMLBroker1: TXMLBroker
  ProviderName = 'ProviderCustomer'
  RemoteServer = DCOMConnection1
  WebDispatch.PathInfo = 'XMLBroker1'
end
object InetXPageProducer1: TInetXPageProducer
  IncludePathURL = '/jssource/'
  object DataForm1: TDataForm
    object LayoutGroup1: TLayoutGroup
      DisplayColumns = 2
      object DataNavigator1: TDataNavigator
        XMLComponent = FieldGroup1
        object FirstButton1: TFirstButton
          XMLComponent = FieldGroup1
          Caption = '|<'
        end
        object PriorButton1: TPriorButton
          XMLComponent = FieldGroup1
          Caption = '<'
        end
        object NextButton1: TNextButton
          XMLComponent = FieldGroup1
          Caption = '>'
        end
        ...
        object ApplyUpdatesButton1: TApplyUpdatesButton
          Caption = 'Apply Updates'
          XMLBroker = XMLBroker1
          XMLUseParent = True
        end
      end
      object FieldGroup1: TFieldGroup
        XMLBroker = XMLBroker1
        object CustNo: TFieldText
          DisplayWidth = 10
          Caption = 'CustNo'
          FieldName = 'CustNo'
        end
        object Company: TFieldText
          DisplayWidth = 30
          Caption = 'Company'
          FieldName = 'Company'
        end
        ...
      end
      object DataNavigator2: TDataNavigator
        XMLComponent = DataGrid1
        object FirstButton2: TFirstButton
          XMLComponent = DataGrid1
          Caption = '|<'
        end
        object PriorPageButton1: TPriorPageButton
          XMLComponent = DataGrid1
          Caption = '<<'
        end
        ...
      end
      object DataGrid1: TDataGrid
        XMLBroker = XMLBroker1
        XMLDataSetField = 'TableOrders'
        DisplayRows = 8
        object OrderNo: TTextColumn
          DisplayWidth = 10
          Caption = 'OrderNo'
          FieldName = 'OrderNo'
        end
        object SaleDate: TTextColumn
          DisplayWidth = 18
          Caption = 'SaleDate'
          FieldName = 'SaleDate'
        end
        ...
      end
    end
  end
end
```

Once the structure is set up, you can deploy the CGI executable on the Web server and see the effect illustrated in Figure 23.14 directly in a browser. Notice that the HTML you receive is rather large, as it includes the entire master/detail structure. Once you've received it, however, you can browse the master table and the detail grid without having to ask the server for any more data.

FIGURE 23.14:

A master/detail relationship displayed in a browser by the IeMd example


http://localhost/scripts/iemd.exe - Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/scripts/iemd.exe
Go
|< < > >| Undo Apply Updates
|< << < > >> >|
Done
Local intranet

CustNo 1356

Company Tom Sawyer Diving Centre

Addr1 632-1 Third Frydenhoj

City Christiansted

State St. Croix

Zip 00820

Country US Virgin Islands

Phone 504-798-3022

TaxRate 0

Contact Chris Thomas

Updated □

OrderNo	SaleDate	EmpNo	PO	Terms	ItemsTotal	AmountPaid
1005	04/20/1988 00:00:00	110		FOB	4807	4807
1059	02/24/1989 00:00:00	109		FOB	2150	2150
1072	04/11/1989 00:00:00	29		Net 30	3596	3596
1080	05/05/1989 00:00:00	45		Net 30	9634	9634
1105	07/21/1992 00:00:00	28		FOB	31219.95	31219.95
1180	08/06/1994 00:00:00	144		Net 30	3640	3640

Obviously, much more could be said about the capabilities of the Internet Express to build a Web front end for a DataSnap server, as a possible alternative to the server-side scripting offered by WebSnap. Server-side scripting certainly allows for wider support of browsers, although most browsers have the minimal JavaScript required by Internet Express. Also, you should consider cases in which you prefer to have immediate feedback on the server for every action of the user or when you prefer to let the user prepare a large delta packet, even working in a disconnected situation, and then receive the entire batch of updates at once.

Using XSLT

Another approach for generating HTML starting from an XML document is the use of the Extensible Stylesheet Language (XSL) or, to be more precise, its XSL Transformations (XSLT) subset. XSLT uses other XML technologies, notably XPath and XPointer to identify portions of documents.

XPath defines a set of rules to locate one or more nodes within a document. The rules are based on a path-lie structure of the node within the XML tree, so that /books/book identifies any *book* node under the *books* document root. XPath uses a few special symbols to identify nodes:

- An asterisk (*) stands for any node; for example, book/* indicates any subnode under a *book* node.
- A dot (.) stands for the current node.
- The pipe symbol (|) indicates alternatives, as in book|ebook.
- A double slash (//) stands for any path, so that //title indicates all of the title nodes, whatever their parent nodes, and books//author indicates any author node under a books node regardless of the nodes in between.
- The at sign (@) indicates an attribute instead of a node, as in the hypothetical author/@lastname. A similar notation can be used to choose only nodes having a given value for an attribute—for example, all authors with a given first name: author[@name="marco"].

There are many other cases, but this short introduction to the rules of XPath should at least get you started and help you understand the following examples. An XSTL document is an XML document that works on the structure of a source XML document and generates in output another XML document, possibly an XHTML document you can view within a Web browser.

NOTE Commonly used XSLT processors include MS-XML, Xalan from the Apache XML project (xml.apache.org), and the Java-based Xt of James Clarke.

The structure of an XSL file is quite simple, although its content can become extremely complex to understand. At the root should be a node like:

```
<xsl:stylesheet version="1.0" xmlns:xsl="...">
```

This node is followed by some of the base commands, such as the definition of a template to operate on a given type of nodes (xsl:template match), the activation of a template for a given node (xsl:apply-templates select), or the extraction of a value from an XML document (xsl:value-of select). There are also specific instructions you can use, including xsl:for-each, xsl:if, xsl:choose, xsl:sort (not available in MSXML the last time I checked), and xsl:number.

XSTL in Practice

After this short and probably unclear explanation, let me discuss a couple of examples. As a starting point, you should study XSL by itself, and then focus on its activation from within a Delphi application.

For your initial tests, you can connect an XSL file directly into an XML file. As you load the XML file in Internet Explorer, this will show you the resulting transformation, usually the HTML. The connection is indicated in the heading of the XML document with a command like:

```
<?xml-stylesheet type="text/xsl" href="sample1embedded.xsl"?>
```

This is what I've done in the `sample1embedded.xml` file available in the XslEmbed project. The related XSL embeds various XSL snippets that I cannot discuss in detail. For example, it grabs the entire list of authors or filters a specific group of them with this code:

```
<h2>All Authors</h2>
<ul>
  <xsl:for-each select="books//author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
<h3>E-Authors</h3>
<ul>
  <xsl:for-each select="books/ebook/author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
```

Some rather more complex code is used to extract nodes only when a specific value is present in a subnode or attribute, regardless of the higher-level nodes. This final XSL snippet also has an `if` statement and produces an attribute in the resulting node, as a way to build an `href` hyperlink in the HTML:

```
<h3>Marco's works (books + ebooks)</h3>
<ul>
  <xsl:for-each select="books/*[author = 'Cantu']">
    <li> <xsl:value-of select="title"/>
        <xsl:if test="url">
           (<a><xsl:attribute name="href"><xsl:value-of select="url"/>
               </xsl:attribute>Jump to document</a>)
        </xsl:if>
     </li>
  </xsl:for-each>
</ul>
```

Again, I suggest you get some more documentation on the topic, but these examples should at least get you started.

XSLT with WebSnap

Within the code of a program, you can execute the `TransformNode` method of a DOM node, passing to it another DOM hosting the XSL document. Instead of using this low-level approach, however, we can let WebSnap help us to create an XSL-based example. In fact, you can create a new WebSnap application (I've built a CGI program called XslCust in this case) and choose an XSLPageProducer component for its main page, to let Delphi help you start with the application code. Actually, Delphi also includes a skeleton XSL file for manipulating a ClientDataSet data packet and adds to the editor many new views. The XSL Text replaces the HTML file; the XML Tree shows the data, if any; the XSL tree shows the XSL within the Internet Explorer ActiveX; the HTML result shows the code produced by the transformation; and (finally) the Preview page shows what a user will see in a browser.

To make this actually work, you must provide some data to the XSLPageProducer component, via its `XMLData` property. This property can be hooked up to an XMLDocument but also directly to an XMLBroker component, as I've done in this case. The broker takes its data from a provider connected to a local table, attached to the Customers table of the classic DBDEMOS.

The effect is that, with the following Delphi-generated XSL, you get (even at design time) the output of Figure 23.15.

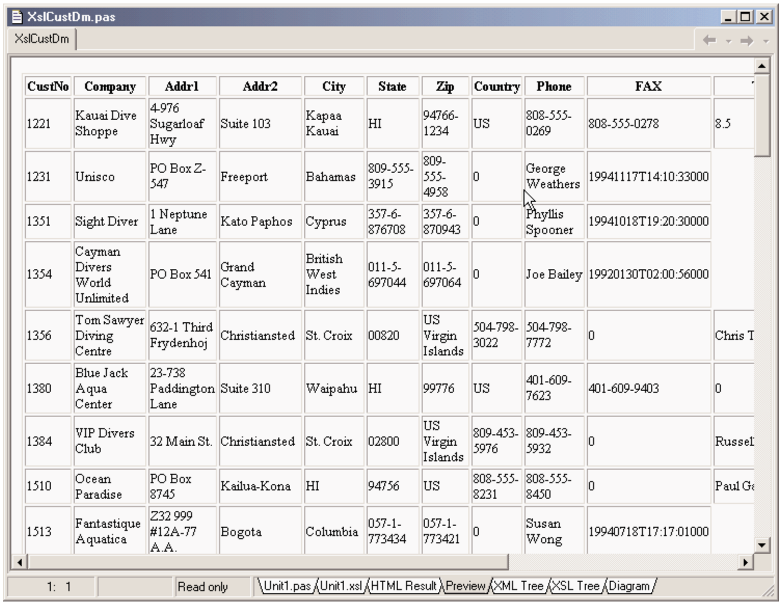


FIGURE 23.15:

The result of an XSTL transformation generated (even at design time) by the XSLPageProducer component in the XslCust example

CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Phone	FAX	
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	808-555-0269	808-555-0278	8.5
1231	Unisco	PO Box Z-547	Freeport	Bahamas	809-555-3915	809-555-4958	0	George Weathers	19941117T14:10:33000	
1351	Sight Diver	1 Neptune Lane	Kato Paphos	Cyprus	357-6-876708	357-6-870943	0	Phyllis Spooner	19941018T19:20:30000	
1354	Cayman Divers World Unlimited	PO Box 541	Grand Cayman	British West Indies	011-5-697044	011-5-697064	0	Joe Bailey	19920130T02:00:56000	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	Christiansted	St. Croix	00820	US Virgin Islands	504-798-3022	504-798-7772	0	Chris T
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310	Waipahu	HI	99776	US	401-609-7623	401-609-9403	0
1384	VIP Divers Club	32 Main St.	Christiansted	St. Croix	02800	US Virgin Islands	809-453-5976	809-453-5932	0	Russel
1510	Ocean Paradise	PO Box 8745	Kailua-Kona	HI	94756	US	808-555-8231	808-555-8450	0	Paul Ga
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	Bogota	Columbia	057-1-773434	057-1-773421	0	Susan Wong	19940718T17:17:01000	

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>

  <xsl:template match="DATAPACKET">
    <table border="1">
    <xsl:apply-templates select="METADATA/FIELDS"/>
    <xsl:apply-templates select="ROWDATA/ROW"/>
    </table>
  </xsl:template>

  <xsl:template match="FIELDS">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

  <xsl:template match="FIELD">
    <th><xsl:value-of select="@attrname"/></th>
  </xsl:template>

  <xsl:template match="ROWDATA/ROW">
    <tr><xsl:for-each select="@*">
      <td><xsl:value-of/></td>
    </xsl:for-each></tr>
  </xsl:template>
</xsl:stylesheet>
```

This code, based heavily on XSL templates, generates an HTML table made of the expansion of field metadata and row data. The fields are used to generate the table heading, with a `<th>` cell for each entry in a single row. The row data is used to fill in the other rows of the table with the value of each attribute (`select="@*"`). At this point, you should be able to modify this XSL file and change the output of the program.

Direct XSL Transformations with the DOM

Using the XSLPageProducer can certainly be handy, but generating multiple pages based on the same data just to handle different possible XSL styles with WebSnap doesn't seem to be the best approach. I've rather built a plain CGI application, called CdsXstl, that can transform a ClientDataSet data packet into different types of HTML, depending on the name of the XSL file passed as parameter to it. The advantage is that I can not only modify but even add new XSL files to the system without having to recompile the program.

To obtain the XSL transformation, the program loads both the XML and the XSL files into two XMLDocument components, called `xmlDom` and `XslDom`. Then it invokes the `transformNode` method of the XML document, passing the XSL document as parameter and filling in a third XMLDocument component, called `HtmlDom`:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  xslfile: string;
  attr: IDOMAttr;
begin
  // open the client dataset and load its XML in a DOM
  ClientDataSet1.Open;
  XmlDom.Xml.Text := ClientDataSet1.XMLData;
  XmlDom.Active := True;
  // load the requested xsl file
  xslfile := Request.QueryFields.Values ['style'];
  if xslfile = '' then
    xslfile := 'customer.xsl';
  xslDom.LoadFromFile ('c:\websites\xsl\' + xslfile);
  XSLDom.Active := True;
  if xslfile = 'single.xsl' then
  begin
    attr := xslDom.DOMDocument.createAttribute('select');
    attr.value := '//ROW[@CustNo="' + Request.QueryFields.Values ['id'] + '"]';
    xslDom.DOMDocument.getElementsByTagName ('xsl:apply-templates').
      item[0].attributes.setNamedItem(attr);
  end;
  // do the transformation
  HTMLDom.Active := True;
  xmlDom.DocumentElement.transformNode (xslDom.DocumentElement, HTMLDom);
  Response.Content := HTMLDom.XML.Text;
end;
```

The code uses the DOM to modify the XSL document for displaying a single record, adding the XPath statement for selecting the record indicated by the `id` query field. This `id` is added to the hyperlink by the XSL with the list of records, but here I'd rather skip listing more XSL files. You can study them, as they are available in the XSL subfolder of the folder for this example.

WARNING To run this program, the XSL files should be deployed to the folder `c:\websites\xsl\`, or anywhere you like after fixing the source code accordingly.

Web Services

Of all of the new features of Delphi 6, one stands out quite clearly: the support for Web services built into the product. The fact I'm discussing it at the end of the book has nothing to do with its importance, but only with the logical flow of the text, and with the fact that this is indeed not the starting point to learn Delphi programming.

But what is a Web service? It is a rapidly emerging technology that has the potential to change the way the Internet works for businesses. Browsing Web pages to enter your orders is fine for individuals (so-called B2C or business-to-consumer applications) but not for companies (so-called B2B or business-to-business applications). If you want to buy a few books, going to a book vendor Web site and punching in your requests is probably fine. But if you run a bookstore and want to place hundreds of orders a day, this is far from a good approach, particularly if you have a program that helps you track your sales and determine reorders. Grabbing the output of this program and reentering it into another application is really ridiculous.

The idea of Web services is to solve this issue: The program used to track sales can automatically create a request and send it to a Web service, which can immediately return information about the order. The next step might be to ask for a tracking number for the shipment. At this point, your program can use another Web service to track the shipment until it is at its destination, so you can tell your customers how long they have to wait. As the shipment arrives, your program can send a reminder via SMS or pager to the people with pending orders, issue a payment with a bank Web service, and … I could continue but I think I've given you an idea. Web services are for computer interoperability as much as the Web and email let people interact.

SOAP and WSDL

If the idea of Web services should be clear by now, what makes them possible is the Simple Object Access Protocol (SOAP). SOAP is built over standard HTTP, so that a Web server can handle the SOAP requests and the related data packets can pass though firewalls. SOAP defines an XML-based notation for requesting the execution of a method by an object on the server, passing parameters to it, and a notation to define the format of a response.

NOTE SOAP was originally developed by DevelopMentor (the training company of COM expert Don Box) and Microsoft, to overcome weaknesses of using DCOM inside Web servers. Submitted to the W3C for standardization, it is being embraced by many companies, with a particular push from IBM. It is too early to see whether there will be a real standardization to let software programs from Microsoft, IBM, Sun, Oracle, and many others truly interoperate or whether some of these vendors will try to push a private version of the standard. In any case, SOAP is a cornerstone of Microsoft's dotNet architecture but also of the current proposals by Sun and Oracle.

SOAP is going to replace COM invocation, at least between different types of computers. Similarly, the definition of a SOAP service in the Web Services Description Language (WSDL) format is going to replace the IDL and type libraries used by COM and COM+. WSDL documents are another type of XML document that provides the metadata definition of a SOAP request. As you get a file in this format (generally published to define a service), you'll be able to create a program to call it.

Specifically, Delphi 6 provides a bidirectional mapping between WSDL and interfaces. This means you can grab a WSDL file and generate an interface for it. At this point, you can create a client program embedding SOAP requests via these interfaces and use a special Delphi component that allows you to convert your local interface requests into SOAP calls (as I doubt you want to manually generate the XML required for a SOAP request).

The other way around, you can define an interface (or use an existing one) and let a Delphi component generate a WSDL description for it. Another component provides you with a SOAP-to-Pascal mapping, so that by embedding this component and an object implementing the interface within a server-side program, you can have your Web service up and running in matter of minutes.

BabelFish Translations

As a first example of the use of Web service, I'm going to build a simple client for the BabelFish translation service offered by AltaVista. Because this is an experimental service, like most others, there is no guarantee that the service will be working by the time you read this. You can find this and other services for experiments on the XMethods Web site (`www.xmethods.com`) and also look for sample Web services provided by my own site (`www.marcocantu.com`).

After downloading the WSDL description of this service from XMethods (also available on the CD-ROM), I invoked Delphi's Web Services Importer in the Web Services page of the New items dialog box and selected the file. Delphi generated an Object Pascal interface for the Web service, as follows:

```
unit babelintf;

interface

uses
  Types, XSBuiltIns;

type
  BabelFishPortType = interface(IInvokable)
    ['{DF96B8F8-DD8E-43A1-9276-4F821D9EA3FA}']
    function BabelFish(const translationmode: String;
      const sourcedata: String): String; stdcall;
  end;

implementation

uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType), '', '');
end.
```

Notice first that the interface inherits from the IInvokable interface. This doesn't add anything in terms of methods to the IInterface base interface of Delphi, but is compiled with the flag used to enable RTTI generation, {$M+}, like the TPersistent class. In this code, you can also see that the interface is registered in the global invocation registry (or InvRegistry), passing the type information reference of the interface type.

NOTE Having RTTI information for interfaces is actually the most important technological advance underlying SOAP invocation. Not that SOAP-to-Pascal mapping isn't important, as it is crucial to simplify the process, but having RTTI for an interface is what makes the entire architecture powerful and robust.

Once you have converted a WSDL definition into an easy-to-use interface, you need a component translating from interface call to SOAP call, and also handling the response and possible errors. This role can be played by the HTTPRio component, which implements the idea of a Remote Invocation Object (RIO) over HTTP. Delphi 6 was built opening up the road for SOAP, but also keeping it open for other remote invocation mechanisms.

In the BabelFish example, the HTTPRio component has the following settings, obtained by choosing the WSDL file first and then selecting the only available service and port from it, directly in the drop-down lists of the Object Inspector:

```
object HTTPRIO1: THTTPRIO
  WSDLLocation = 'C:\md6code\23\BabelFish\BabelFishService.xml'
  Service = 'BabelFish'
  Port = 'BabelFishPort'
  HTTPWebNode.Agent = 'Borland SOAP 1.1'
  Converter.Options = [soSendMultiRefObj, soTryAllSchema]
end
```

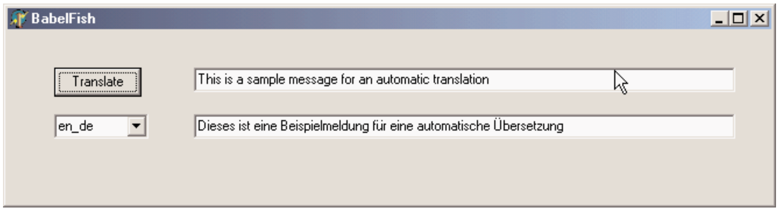
At this point, there is very little left to do. We have information about the service that can be used for its invocation, and we know the parameters required by the only available method. The two elements are merged by extracting the interface you want to call directly from the HTTPRio component, with an expression like HTTPRIO1 as BabelFishPortType. It might seem rather astonishing at first, but it is also outrageously simple. This is the Web service call done by the example:

```
EditTarget.Text := (HTTPRIO1 as BabelFishPortType).
  BabelFish(ComboBoxType.Text, EditSource.Text);
```

The resulting output of the program, depicted in Figure 23.16, allows you to learn foreign languages (although the teacher here has its shortcomings!). I haven't replicated the same example with stock options, currencies, weather forecasts, and the many other services available, as they would look much the same.

FIGURE 23.16:

An example of a translation from English to German obtained by AltaVista's BabelFish via a Web service



This is a sample message for an automatic translation

Dieses ist eine Beispielmeldung für eine automatische Übersetzung

Building a Web Service

If calling a Web service in Delphi 6 is very straightforward, the same can be said of the development of an actual service. If you go into the Web Services page of the New items dialog box, you can see the SOAP Server Application option. Selecting it, Delphi presents you a list quite similar to selection of a WebBroker application. A Web service, in fact, is typically hosted by a Web server, using one of the available Web server extension technologies (CGI, ISAPI, Apache modules, etc.). After completing this step, Delphi will add three components to the resulting Web module, which is just a plain Web module, with no special additions:

- The HTTPSoapDispatcher component has the role of receiving the Web request, as any other HTTP dispatcher does.
- The HTTPSoapPascalInvoker component does the reverse operation of the HTTPRio component, as it can translate SOAP requests into calls of Pascal interfaces (instead of shifting interface method calls into SOAP requests).
- The WSDLHTMLPublish component can be used to extract the WSDL definition of the service from the interfaces it support, and performs the opposite role of the Web Services Importer Wizard. Technically, this is another HTTP dispatcher.

Once this framework is in place—something you can also do by adding the three components above to an existing Web module—we can start writing a service. As an example I've taken the euro conversion example of Chapter 4, "The Run-Time Library," and transformed it into a Web service called ConvertService. First of all, I've added to the program a unit defining the interface of the service, as follows:

```
type
  IConvert = interface(IInvokable)
  ['{FF1EAA45-0B94-4630-9A18-E768A91A78E2}']
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

Defining an interface directly in code, without having to use a tool such as the Type Library Editor, provides a great advantage. Notice that I've given a GUID to the interface, as usual, and used the `stdcall` calling convention, as the SOAP converter does not support the default `register` calling convention.

In the same unit defining the interface of the service, we should also register it, an operation which will be useful on both the client and server sides of the program, as we will be able to include this interface definition unit in both.

```
uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IConvert));
```

Now that we have an interface we can expose to the public, we have to provide an implementation for it, again by means of the standard Pascal code (and with the help of the predefined `TInvokableClass` class:

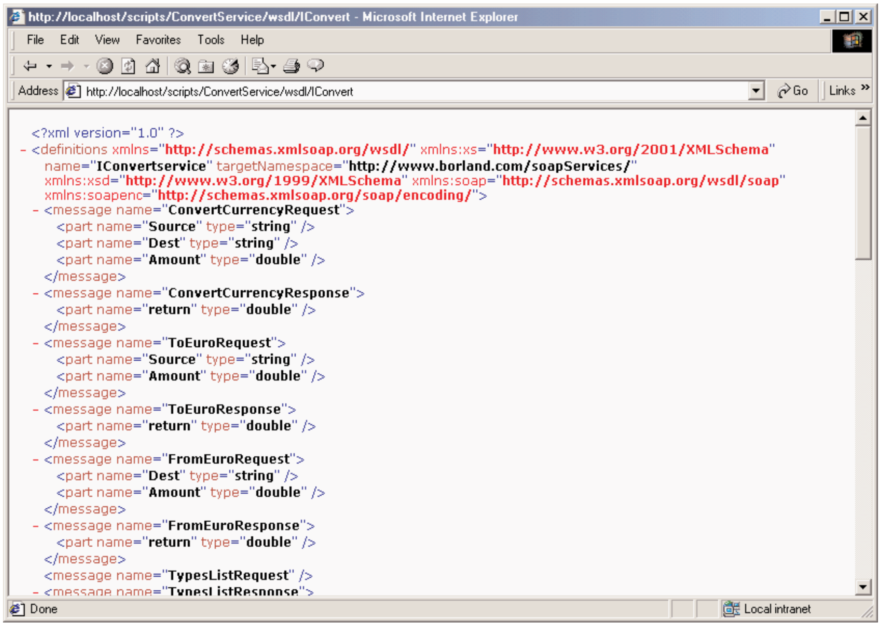
```
type
  TConvert = class (TInvokableClass, IConvert)
  protected
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

The implementation of these functions, which call in the code of the euro conversion system of Chapter 4, is not discussed here as it has little to do with the development of the service itself. What is important to notice, instead, is that this implementation unit also has a registration call in its initialization section:

```
InvRegistry.RegisterInvokableClass (TConvert);
```

This is basically all. By registering the interface, we'll make it possible for the program to generate a WSDL description, as you can see in Figure 23.17, where I've used a browser to connect to the *wsdl* action of the service, implemented by the WSDLHTMLPublish component.

FIGURE 23.17: The WSDL file generated automatically from the interface published by the ConvertService program


http://localhost/scripts/ConvertService/wsdl/IConvert - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/scripts/ConvertService/wsdl/IConvert
Go
Links
Done
Local intranet

```
  <?xml version="1.0" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    name="IConvertservice" targetNamespace="http://www.borland.com/soapServices/"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
  - <message name="ConvertCurrencyRequest">
      <part name="Source" type="string" />
      <part name="Dest" type="string" />
      <part name="Amount" type="double" />
    </message>
  - <message name="ConvertCurrencyResponse">
      <part name="return" type="double" />
    </message>
  - <message name="ToEuroRequest">
      <part name="Source" type="string" />
      <part name="Amount" type="double" />
    </message>
  - <message name="ToEuroResponse">
      <part name="return" type="double" />
    </message>
  - <message name="FromEuroRequest">
      <part name="Dest" type="string" />
      <part name="Amount" type="double" />
    </message>
  - <message name="FromEuroResponse">
      <part name="return" type="double" />
    </message>
    <message name="TypesListRequest" />
  - <message name="TypesListResponse">
```

Of course, you cannot call the service from a browser, as the role of a Web service is not to display data on the Web but rather to let applications interoperate (still, you can reasonably have a Web application calling a service). Before I discuss the client application, though, let me cover another interesting element. For debugging purposes, I've added to the Web module an actual action, connected with code to generate information about the registered interfaces and servers:

```
Response.Content :=
  '<h3>GetMethExternalName - ToEuro</h3><p>' +
  InvRegistry.GetMethExternalName(TypeInfo(IConvert), 'ToEuro</p>') +
  '<h3>GetInterfaceExternalName - IConvert</h3><p>' +
  InvRegistry.GetInterfaceExternalName(TypeInfo(IConvert)) + '</p>' +
  '<h3>GetNamespaceByGUID - IConvert</h3><p>' +
  InvRegistry.GetNamespaceByGUID (IConvert) + '</p>';
```

The first call verifies whether a given method is properly registered and really exists, by calling the `GetMethExternalName` of the invocation registry. The result, unsurprisingly, is the same string passed as parameter, ToEuro. The second call, `GetInterfaceExternalName`, should return the external name of the interface, but I haven't been able to make it work properly (I left it in anyway, as it is supposed to work). The last call, GetNamespaceByGUID, returns the XML namespace of the interface, urn:ConvertIntf-IConvert. There are other similar calls you can make against the registry, which are quite interesting and demonstrate the power of this approach and of RTTI for interfaces.

Having said this, let me move to the client application, calling the service. This time I don't really need to start from the WSDL file, as I already have the Pascal interface. This time the form doesn't even have the HTTPRio component, which is created in code:

```
private
  Invoker: THTTPRio;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Invoker := THTTPRio.Create(nil);
  Invoker.URL := 'http://localhost/scripts/ConvertService.exe/soap/iconvert';
  ConvIntf := Invoker as IConvert;
end;
```

As an alternative to using a WSDL file, the SOAP invoker component can be associated with an URL. Once this association has been done and the required interface has been extracted from the component, you can start writing straight Pascal code to invoke the service, as we saw earlier.

A user can fill the two combo boxes, calling the TypesList method, which returns a list of available currencies within a string (separated by semicolons). This list is extracted by replacing any semicolon with a line break and then assigning the multiline string directly to the combo items:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  TypeNames: string;
begin
  TypeNames := ConvIntf.TypesList;
  ComboBoxFrom.Items.Text := StringReplace (TypeNames, ';', sLineBreak,
    [rfReplaceAll]);
  ComboBoxTo.Items := ComboBoxFrom.Items;
end;
```

At this point, after selecting two currencies, you can perform the conversion, with this code and the result of Figure 23.18:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  LabelResult.Caption := Format ('%n', [(ConvIntf.ConvertCurrency(
    ComboBoxFrom.Text, ComboBoxTo.Text, StrToFloat(EditAmount.Text)))]);
end;
```

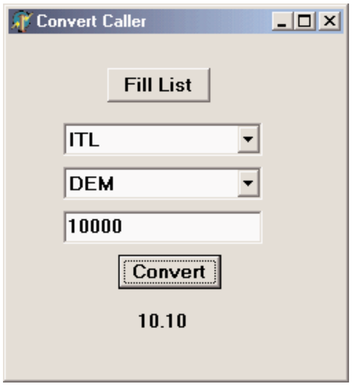


FIGURE 23.18:

The ConvertCaller client of the ConvertService Web service shows how few German marks you used to get for so many Italian liras, before the euro changed all this.

DataSnap over SOAP

Now that we have a reasonably good idea of how to build a SOAP server and a SOAP client, we can have a look at how to use this technology in building a multitier DataSnap application. We'll use a Soap Server Data Module to create the new Web service and the SoapConnection component to connect a client application to it.

Let's look at the server side first. You have to move to the Web Services page of the New Items dialog box and use the Soap Server Application icon first to create a new Web service, and then use the Soap Server Data Module icon to add a DataSnap server-side data module to the SOAP server. This is what I've done in the SoapDataServer example (which uses the Web App Debugger architecture for testing purposes). From this point on, all you do is write a *normal* DataSnap server (or actually a middle-tier DataSnap application) as discussed in Chapter 17. In this specific case, I've added to the program InterBase access by means of dbExpress, resulting in the following structure:

```
object SoapTestDm: TSoapTestDm
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
  end
end
```

The data module built for a SOAP-based DataSnap server defines a custom interface (so you can add methods to it) inheriting from `IAppServer`, which is now defined as a published interface (even though it doesn't inherit from `IInvokable`). The implementation class, `TSoapTestDm`, is the data module itself, as in other DataSnap types of servers. This is the code Delphi generated for me:

```
type
  ISoapTestDm = interface(IAppServer)
    ['{1F109687-6D8B-4F85-9BF5-EFFC87A9F10F}']
  end;

  TSoapTestDm = class(TSoapDataModule, ISoapTestDm, IAppServer)
    DataSetProvider1: TDataSetProvider;
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
  end;
```

The base `TSoapDataModule` doesn't inherit from `TInvokableClass`. This is not a problem as long as you provide an extra procedure to create the object (which is what `TInvokableClass` does for you) and add it to the registration code:

```
procedure TSoapTestDmCreateInstance(out obj: TObject);
begin
  obj := TSoapTestDm.Create(nil);
end;

initialization
  InvRegistry.RegisterInvokableClass(TSoapTestDm, TSoapTestDmCreateInstance);
  InvRegistry.RegisterInterface(TypeInfo(ISoapTestDm));
```

The server application actually also publishes the `IAppServer` interface, thanks to the only line of code in the SOAPMidas unit.

WARNING Web service applications should not include more than one SOAP Data Module, as the registration cannot distinguish between multiple implementations of the same `IAppServer` interface.

To build the client application, called SoapDataClient, I've started with a plain program and added a SoapConnection component to it (from the Web Services page of the palette), hooking it to the URL of the DataSnap Web service, referring to the specific interface we are looking for:

```
object SoapConnection1: TSoapConnection
  Agent = 'Borland SOAP 1.1'
  URL = 'http://localhost:1024/SoapDataServer.SoapDataServer/Soap/ISoapTestDm'
end
```

From this point on, I've proceeded as usual, adding a ClientDataSet component, a DataSource, and a DBGrid to the program, choosing the only available provider for the client dataset, and hooking the rest as usual. Not surprisingly, for this simple example, the client application has little custom code: a single call to open the connection when a button is clicked (to avoid startup errors) and an `ApplyUpdates` call to send changes back to the database.

Regardless of the apparent similarity of this program to all of the other DataSnap client and server programs built in Chapter 17, there is a very important difference worth underlining: The SoapDataServer and SoapDataClient programs do not use COM for exposing or calling the `IAppServer` interface. Quite the opposite—the socket- and HTTP-based connections of DataSnap still rely on local COM objects and a registration of the server in the Windows Registry. The native SOAP-based support provided by Delphi 6, instead, allows for a totally custom solution independent from COM and with many more chances to be ported to other operating systems (Linux being certainly the first, with a future release of Kylix).

What's Next?

In this final chapter of the book, I've covered XML and related technologies, including XSLT, SOAP, WSDL, XML schemas, XPath, and a few more. We've seen how Delphi 6 provides simplified DOM programming, XML access using interfaces, and XML transformations. I've also covered Internet Express and the development of Web services.

Besides tracking what goes on in the area of SOAP and WSDL, particularly in terms of standards conformance by the major players, there are a few interesting initiatives you should probably keep track of if you're interested in the development in business-to-business services. One of them is the UDDI proposal (`www.uddi.org`), pushed by Microsoft, IBM, Ariba, and many other companies, to create a universal registry of services. Another is ebXML (`www.ebxml.org`), a proposal by the U.N. office that defined the EDI standards for an XML-based global business exchange.

Of course, I don't want to delve too much into these nontechnical issues, but I thought it was worth mentioning them at the end of this book, as I try to give a few hints at a sort of "what's next" for Delphi programmers. Delphi is indeed a strong player in both the Windows and Linux client markets, in the client/server and enterprise application markets, and now takes a bold step in the areas of Web development and Web services.

Just as Borland wants to provide the best tools to developers, I hope this book has helped you master Delphi, the most successful tool Borland has brought to the market in the last few years. Remember to check from time to time the reference, foundations, and advanced material I've collected on my Web site (`www.marcocantu.com`). Check my site also for eventual updates and integration of the material in the book, and feel free to use the newsgroups hosted there for your questions about the book and about Delphi in general. Much of this material could not be included in the book, simply because of space constraints. Some of this extra material is actually already available on the companion CD, where you can continue reading about other aspects of Delphi programming.

Developer’s Guide

Borland®

Delphi™ 6

for Windows

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Refer to the DEPLOY document located in the root directory of your Delphi 6 product for a complete list of files that you can distribute in accordance with the Delphi 6 License Statement and Limited Warranty.

Inprise may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

Printed in the U.S.A.

HDE1350WW21001 1E0R0501
0102030405-9 8 7 6 5 4 3 2 1
D3

Contents

Chapter 4
Common programming tasks 4-1

Chapter 5
Building applications, components, and libraries 5-1

Chapter 6
Developing the application user interface 6-1

Chapter 7 Working with controls 7-1

Chapter 8
Working with graphics and multimedia 8-1

Chapter 9
Writing multi-threaded applications 9-1

Chapter 10
Using CLX for cross-platform development 10-1

Chapter 11
Working with packages and components 11-1

Chapter 12
Creating international applications 12-1

Chapter 13
Deploying applications 13-1

Part II
Developing database applications

Chapter 14
Designing database applications 14-1

Chapter 15
Using data controls 15-1

Chapter 16
Using decision support components 16-1

Chapter 17

Connecting to databases 17-1

Chapter 18

Understanding datasets 18-1

Chapter 19
Working with field components 19-1

Chapter 20
Using the Borland Database Engine 20-1

Chapter 21
Working with ADO components 21-1

Chapter 22
Using unidirectional datasets 22-1

Chapter 23
Using client datasets 23-1

Chapter 24
Using provider components 24-1

Chapter 25
Creating multi-tiered applications 25-1

Part III
Writing Internet applications

Chapter 27
Creating Internet applications 27-1

Chapter 28
Using Web Broker 28-1

Chapter 29 Using WebSnap 29-1

Chapter 30
Working with XML documents 30-1

Chapter 31
Using Web Services 31-1

Chapter 32
Working with sockets 32-1

Part IV
Developing COM-based applications

Chapter 33
Overview of COM technologies 33-1

Chapter 34
Working with type libraries 34-1

Chapter 35
Creating COM clients 35-1

Part V
Creating custom components

Chapter 43
Creating events 43-1

Chapter 44
Creating methods 44-1

Chapter 45
Using graphics in components 45-1

Chapter 46
Handling messages 46-1

Chapter 47
Making components available at design time 47-1

Chapter 48
Modifying an existing component 48-1

Chapter 49
Creating a graphic component 49-1

Chapter 50
Customizing a grid 50-1

Chapter 51
Making a control data aware 51-1

Chapter 52
Making a dialog box a component 52-1

Index I-1

Tables

Figures

Chapter

1

Introduction

The *Developer's Guide* describes intermediate and advanced development topics, such as building client/server database applications, writing custom components, and creating Internet Web server applications. It allows you to build applications that meet many industry-standard specifications such as SOAP, TCP/IP, COM+, and ActiveX. Many of the advanced features that support Web development, advanced XML technologies, and database development require components or wizards that are not available in all versions of Delphi.

The *Developer's Guide* assumes you are familiar with using Delphi and understand fundamental Delphi programming techniques. For an introduction to Delphi programming and the integrated development environment (IDE), see the Quick Start manual or the online Help.

What's in this manual?

This manual contains five parts, as follows:

- **Part I, "Programming with Delphi,"** describes how to build general-purpose Delphi applications. This part provides details on programming techniques you can use in any Delphi application. For example, it describes how to use common Visual Component Library (VCL) or Component Library for Cross-platform (CLX) objects that make user interface programming easy. Objects are available for handling strings, manipulating text, implementing common dialogs, and so on. This section also includes chapters on working with graphics, error and exception handling, using DLLs, OLE automation, and writing international applications.

  A chapter describes how to use objects in the Borland Component Library for Cross-Platform (CLX) to develop applications that can be compiled and run on either Windows or Linux platforms.

The chapter on deployment details the tasks involved in deploying your application to your application users. For example, it includes information on effective compiler options, using InstallShield Express, licensing issues, and how to determine which packages, DLLs, and other libraries to use when building the production-quality version of your application.

- **Part II, "Developing database applications,"** describes how to build database applications using database tools and components. Delphi lets you access many types of databases, including local databases such as Paradox and dBASE, and network SQL server databases like InterBase, Oracle, and Sybase. You can choose from a variety of data access mechanisms, including dbExpress, the Borland Database Engine, InterbaseExpress, and ADO. To implement the more advanced database applications, you need the Delphi features that are not available in all versions.
- **Part III, "Writing Internet applications,"** describes how to create applications that are distributed over the internet. Delphi includes a wide array of tools for writing Web server applications, including the Web Broker architecture, which lets you create cross-platform server applications, WebSnap, which lets you design Web pages in a GUI environment, support for working with XML documents, and an architecture for using SOAP-based Web Services. For lower-level support that underlies much of the messaging in Internet applications, this section also describes how to work with socket components. The components that implement many of these features are not available in all versions of Delphi.
- **Part IV, "Developing COM-based applications,"** describes how to build applications that can interoperate with other COM-based API objects on the system such as Windows Shell extensions or multimedia applications. Delphi contains components that support the ActiveX, COM+, and a COM-based library for COM controls that can be used for general-purpose and Web-based applications. Support for COM controls is not available in all editions of Delphi. To create ActiveX controls, you need the Professional or Enterprise edition.
- **Part V, "Creating custom components,"** describes how to design and implement your own components, and how to make them available on the Component palette of the IDE. A component can be almost any program element that you want to manipulate at design time. Implementing custom components entails deriving a new class from an existing class type in the VCL or CLX class libraries.

Manual conventions

This manual uses the typefaces and symbols described in Table 1.1 to indicate special text.

Table 1.1 Typefaces and symbols

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears on screen or in Object Pascal code. It also represents anything you must type.
[ ]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
Boldface	Boldfaced words in text or code listings represent Object Pascal keywords or compiler options.
Italics	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."

Developer support services

Inprise also offers a variety of support options to meet the needs of its diverse developer community. To find out about support offerings for Delphi, refer to http://www.borland.com/devsupport/delphi.

Additional Delphi Technical Information documents and answers to Frequently Asked Questions (FAQs) are also available at this Web site.

From the Web site, you can access many newsgroups where Delphi developers exchange information, tips, and techniques. The site also includes a list of books about Delphi.

Ordering printed documentation

For information about ordering additional documentation, refer to the Web site at shop.borland.com.

Part

I

Programming with Delphi

The chapters in “Programming with Delphi” introduce concepts and skills necessary for creating Delphi applications using any edition of the product. They also introduce the concepts discussed in later sections of the *Developer’s Guide*.

Chapter

2

Developing applications with Delphi

Borland Delphi is an object-oriented, visual programming environment for rapid development of 32-bit applications for deployment on Windows and Linux. Using Delphi, you can create highly efficient applications with a minimum of manual coding.

Delphi provides a comprehensive class library called the *Visual Component Library* (VCL), Borland Component Library for Cross Platform (CLX), and a suite of Rapid Application Development (RAD) design tools, including application and form templates, and programming wizards. Delphi supports truly object-oriented programming:

- the VCL class library includes objects that encapsulate the Windows API as well as other useful programming techniques (Windows)
- the CLX class library includes objects that encapsulate the Qt library (Windows or Linux)

This chapter briefly describes the Delphi development environment and how it fits into the development life cycle. The rest of this manual provides technical details on developing general-purpose, database, Internet and Intranet applications, and includes information on creating ActiveX and COM controls and writing your own components.

Integrated development environment

When you start Delphi, you are immediately placed within the integrated development environment, also called the IDE. This environment provides all the tools you need to design, develop, test, debug, and deploy applications.

Delphi's development environment includes a visual form designer, Object Inspector, Object TreeView, Component palette, Project Manager, source code editor, and debugger among other tools. Some tools may not be included in all versions of the product. You can move freely from the visual representation of an object (in the form designer), to the Object Inspector to edit the initial runtime state of the object, to the source code editor to edit the execution logic of the object. Changing code-related properties, such as the name of an event handler, in the Object Inspector automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the Object Inspector.

The IDE supports application development throughout the stages of the product life cycle—from design to deployment. Using the tools in the IDE allows for rapid prototyping and shortens development time.

A more complete overview of the development environment is presented in the *Quick Start* manual included with the product. In addition, the online Help system provides help on all menus, dialogs, and windows.

Designing applications

Delphi includes all the tools necessary to start designing applications:

- A blank window, known as a *form*, on which to design the UI for your application.
- Extensive class libraries with many reusable objects.
- An Object Inspector for examining and changing object traits.
- A Code editor that provides direct access to the underlying program logic.
- A Project Manager for managing the files that make up one or more projects.
- Many other tools such as an image editor on the toolbar and an integrated debugger on menus to support application development in the IDE.
- Command-line tools including compilers, linkers, and other utilities.

You can use Delphi to design any kind of 32-bit application—from general-purpose utilities to sophisticated data access programs or distributed applications. Delphi's database tools and data-aware components let you quickly develop powerful desktop database and client/server applications. Using Delphi's data-aware controls, you can view live data while you design your application and immediately see the results of database queries and changes to the application interface.

Chapter 5, "Building applications, components, and libraries" introduces Delphi's support for different types of applications.

Many of the objects provided in the class library are accessible in the IDE from the Component palette. The Component palette shows all of the controls, both visual and nonvisual, that you can place on a form. Each tab contains components grouped by functionality. By convention, the names of objects in the class library begin with a T, such as *TStatusBar*.

One of the revolutionary things about Delphi is that you can create your own components using Object Pascal. Most of the components provided are written in Object Pascal. You can add components that you write to the Component palette and customize the palette for your use by including new tabs if needed.

You can also use Delphi for cross platform development on both Linux and Windows by using CLX. CLX contains a set of classes that, if used instead of those in the VCL, allow your program to port between Windows and Linux.

Developing applications

As you visually design the user interface for your application, Delphi generates the underlying Object Pascal code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment as well.

Creating projects

All of Delphi's application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Delphi.

At the top of the project hierarchy, is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the Project Manager. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the Project Manager, the project file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of a Delphi application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Delphi automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

Editing code

The Delphi Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the Object Inspector. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changed and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor and continue adjusting the form from there.

The Delphi code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the VCL objects, CLX objects, RTL sources, all of the Delphi project files can be viewed and edited in the Code editor.

Compiling applications

When you have finished designing your application interface on the form, writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether or not they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to use Build when you've changed global compiler directives, to ensure that all code compiles in the proper state. You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application.

If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose Project | Compile All Projects or Project | Build All Projects with the project group selected in the Project Manager.

Debugging applications

Delphi provides an integrated debugger that helps you find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging.

The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the values of variables, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed. The debugger is described in online Help.

You can also use exception handling to recognize, locate, and deal with errors. Exceptions in Delphi are classes, like other classes in Delphi, except, by convention, they begin with an E rather than the initial T for other classes.

Deploying applications

Delphi includes add-on tools to help with application deployment. For example, InstallShield Express (not available in all versions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. Refer to Chapter 13, “Deploying applications” for specific information on deployment.

Note Not all versions of Delphi have deployment capabilities.

TeamSource software (not available in all versions) is also available for tracking application updates.

Chapter

3

Using the component libraries

This chapter presents an overview of the component libraries and introduces some of the components that you can use while developing applications. Delphi includes both the Visual Component Library (VCL) and the Borland Component Library for Cross-Platform (CLX). The VCL is for Windows development and CLX is for cross-platform development on Windows and Linux. They are two different class libraries but they have many similarities. Objects, properties, methods, and events that are not in CLX are marked “VCL only.”

Understanding the component libraries

VCL and CLX are class libraries made up of objects, some of which are also components or controls, that you use when developing applications. Both libraries look very similar and contain many of the same objects. Some objects in the VCL implement features that are available on Windows only such as objects that appear on the ADO, BDE, QReport, COM+, Web Services, and Servers tabs on the Component palette. Virtually all CLX objects are available on both Windows and Linux.

VCL and CLX objects are active entities that contain all necessary data and the “methods” (code) that modify the data. The data is stored in the fields and properties of the objects, and the code is made up of methods that act upon the field and property values. Each object is declared as a “class.” All VCL and CLX objects descend from the ancestor object *TObject* including objects that you develop in Object Pascal.

A subset of objects are components. Components are objects that you can place on a form or data module and manipulate at design time. Components appear on the Component palette. You can specify their properties without writing code. All VCL or CLX components descend from the *TComponent* object.

Components are objects in the true object-oriented programming (OOP) sense because they

- Encapsulate a set of data and data-access functions
- Inherit data and behavior from the objects they are derived from
- Operate interchangeably with other objects derived from a common ancestor, through a concept called *polymorphism*

Unlike most components, objects do not appear on the Component palette. Instead, a default instance variable is declared in the unit of the object, or you have to declare one yourself.

Controls are a special kind of component that is visible to users at runtime. Controls are a subset of components. Controls are visual components that you can see when your application is running. All controls have properties in common that specify their visual attributes, such as *Height* and *Width.* The properties, methods, and events that all controls have in common are inherited from *TControl*.

Refer to Chapter 10, “Using CLX for cross-platform development” for details about cross-platform programming and the differences between the Windows and Linux environments. Detailed reference material on all of the objects in the VCL or CLX is accessible using online Help while you are programming. From within the code editor, place the cursor anywhere on the object and press F1 to display help on VCL or CLX components.

If you are using Kylix while developing cross-platform applications, Kylix also includes a *Developer’s Guide* that is tailored for the Linux environment. You can refer to the manual both in the Kylix online Help or the printed manual provided with the Kylix product.

Properties, methods, and events

Both the VCL and CLX form hierarchies of objects that are tied to the Delphi IDE, where you can develop applications quickly. The objects in both component libraries are based on properties, methods, and events. Each object includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). The VCL is written in Object Pascal, whereas CLX is based on Qt, a C++ class library.

Properties

Properties are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *Visible* property determines whether an object can be seen or not in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at runtime, you can see and change properties at design time and get immediate feedback as the components change in the IDE.

- Properties can be accessed in the Object Inspector where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.
- The actual calls to get and set the values are methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.
- You can implement logic that triggers events or modifies other data during the access of the property. For example, changing the value of one property may require the modification of another. You can make the change in the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing a one property on one object could effect several objects. For example, setting the *Checked* property on a radio button effects all of the radio buttons in the group.

Methods

A *method* is a procedure that is always associated with a class. Methods define the behavior of an object. Class methods can access all the *public*, *protected,* and *private* properties and data members of the class and are commonly referred to as member functions.

Events

An *event i*s an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform next. They may choose a menu item, click a button, or mark some text. You can write code to handle the events you're interested in, rather than writing code that always executes in the same restricted order.

Regardless of how an event is called, Delphi looks to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

The kinds of events that can occur can be divided into two main categories:

- User events
- System events

Regardless of how the event was called, Delphi looks to see if you have assigned any code to handle that event. If you have, then that code is executed; otherwise, nothing is done.

User events

User events are actions that are initiated by the user. Examples of user events are *OnClick* (the user clicked the mouse), *OnKeyPress* (the user pressed a key on the keyboard), and *OnDblClick* (the user double-clicked a mouse button). These events are always tied to a user's actions.

System events

System events are events that the operating system fires for you. For example, the *OnTimer* event (the Timer component issues one of these events whenever a predefined interval has elapsed), the *OnCreate* event (the component is being created), the *OnPaint* event (a component or window needs to be redrawn), and so on. Usually, system events are not directly initiated by a user action.

Object Pascal and the class libraries

Object Pascal, a set of object-oriented extensions to standard Pascal, is the language of Delphi. Using Delphi's Component palette and Object Inspector, you can place VCL or CLX components on forms and manipulate their properties without writing code.

All objects descend from *TObject*, an abstract class whose methods encapsulate fundamental behavior like construction, destruction, and message handling. *TObject* is the immediate ancestor of many simple classes.

Components in the VCL or CLX descend from the abstract class *TComponent*. Components are objects that you can manipulate on forms at design time. Visual components—that is, components like *TForm* and *TSpeedButton* that appear on the screen at runtime—are called *controls*, and they descend from *TControl*.

In addition to the visual components, the component libraries contain many nonvisual objects. The IDE allows you to add many nonvisual components to your programs by dropping them onto forms. For example, if you were writing an application that connects to a database, you might place a *TDataSource* component on a form. Although *TDataSource* is nonvisual, it is represented on the form by an icon (which doesn't appear at runtime). You can manipulate the properties and events of *TDataSource* in the Object Inspector just as you would those of a visual control.

When you write classes of your own in Object Pascal, they should descend from *TObject* in the class library that you plan to use. Use VCL if you're writing a Windows application or CLX if writing a cross-platform application. By deriving new classes from the appropriate base class (or one of its descendants), you provide your classes with essential functionality and ensure that they work with the other classes in the class library.

Using the object model

Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you create an object (or, more formally, a class), you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

If you want to create new components and put them on the Component palette, see Chapter 40, "Overview of component creation."

What is an object?

An object, or *class*, is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements.

You can begin to understand objects if you understand Object Pascal *records* or *structures* in C. Records are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of VCL and CLX objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

Examining a Delphi object

When you create a new project, Delphi displays a new form for you to customize. In the Code editor, Delphi declares a new class type for the form and produces the code that creates the new form instance. The code generated for a new Windows application looks like this:

```
unit Unit1;
interface

uses Windows, Classes, Graphics, Forms, Controls, Dialogs;

type
  TForm1 = class(TForm){ The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end;{ The type declaration of the form ends here }

var
  Form1: TForm1;

implementation{ Beginning of implementation part }
{$R *.DFM}
end.{ End of implementation part and unit}
```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no fields or methods, because you haven't added to the form any components (the fields of the new object) and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

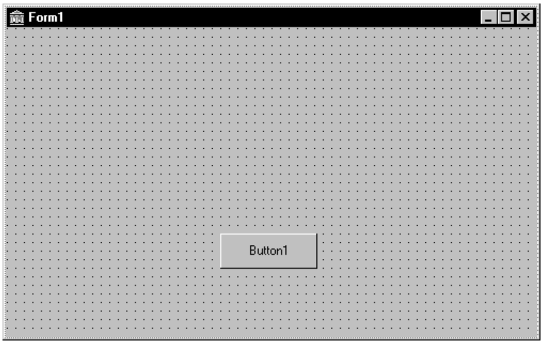
```
var
  Form1: TForm1;
```

Form1 represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete Delphi application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:

Figure 3.1 A simple form



When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls;
```

```
type
  TForm1 = class(TForm)
    Button1: TButton;{ New data field }
    procedure Button1Click(Sender: TObject);{ New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);{ The code of the new method }
begin
  Form1.Color := clGreen;
end;

end.
```

TForm1 has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write in Delphi are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared within the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the **implementation** part of the unit.

Changing the name of a component

You should always use the Object Inspector to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorBox*. When you change the form's *Name* property in the Object Inspector, the new name is automatically reflected in the form's .dfm or .xfm file (which you usually don't edit manually) and in the Object Pascal source code that Delphi generates:

```
unit Unit1;

interface

uses Windows, Classes, Graphics, Forms, Controls;

type
  TColorBox = class(TForm){ Changed from TForm1 to TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

```
  ColorBox: TColorBox;{ Changed from Form1 to ColorBox }

implementation

{$R *.DFM}

procedure TColorBox.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;{ The reference to Form1 didn't change! }
end;

end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorBox.Button1Click(Sender: TObject);
begin
  ColorBox.Color := clGreen;
end;
```

Inheriting data and code from an object

The *TForm1* object described seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the component *TForm*. When you add a new form to your project, you start with *TForm* and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, Delphi automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived object inherits all the properties, events, and methods of the object it derives from. The derived object is called a *descendant* and the object it derives from is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. An object can have only one immediate ancestor, but it can have many direct descendants.

Scope and qualifiers

Scope determines the accessibility of an object's fields, properties, and methods. All members declared within an object are available to that object and its descendants. Although a method's implementation code appears outside of the object declaration, the method is still within the scope of the object because it is declared within the object's declaration.

When you write code to implement a method that refers to properties, methods, or fields of the object where the method is declared, you don't need to preface those identifiers with the name of the object. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

Delphi creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the **uses** clause of *Form1*'s unit.

The scope of an object extends to the object's descendants. You can, however, redeclare a field, property, or method within a descendant object. Such redeclarations either hide or override the inherited member.

For more information about scope, inheritance, and the **uses** clause, see the *Object Pascal Language Guide*.

Private, protected, public, and published declarations

When you declare a field, property, or method, the new member has a *visibility* indicated by one of the keywords **private**, **protected**, **public**, or **published**. The visibility of a member determines its accessibility to other objects and units.

- A private member is accessible only within the unit where it is declared. Private members are often used within a class to implement other (public or published) methods and properties.
- A protected member is accessible within the unit where its class is declared and within any descendant class, regardless of the descendant class's unit.
- A public member is accessible from wherever the object it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.

- A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the Object Inspector at design time.

For more information about visibility, see the *Object Pascal Language Guide*.

Using object variables

You can assign one object variable to another object variable if the variables are of the same type or assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable you are assigning to is an ancestor of the type of the variable being assigned. For example, here is a *TDataForm* type declaration (VCL only) and a variable declaration section declaring two variables, *AForm* and *DataForm*:

```
type
  TDataForm = class(TForm)
  Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Database1: TDatabase;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AForm: TForm;
  DataForm: TDataForm;
```

AForm is of type *TForm*, and *DataForm* is of type *TDataForm*. Because *TDataForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := DataForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
⋮
end;
```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word **is**. For example,

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

Creating, instantiating, and destroying objects

Many of the objects you use in Delphi, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and datasource components, have no visual representation at runtime.

You may want to create your own objects. For example, you could create a *TEmployee* object that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the **interface** or **implementation** part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically by Delphi. But if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a *destructor*) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a **nil** reference before calling *Destroy*. For example,

```
Employee.Free
```

destroys the *Employee* object and deallocates its memory.

Components and ownership

Delphi has a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

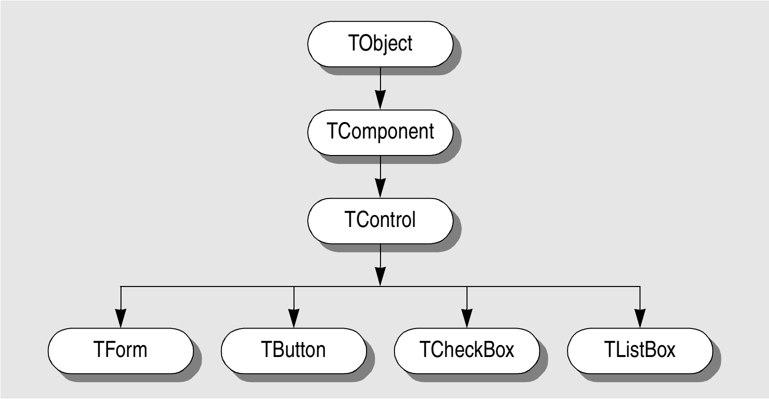
Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

Note Don't confuse a component's *owner* with its *parent*. See "Parent properties" on page 3-19".

Objects, components, and controls

Figure 3.2 is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.

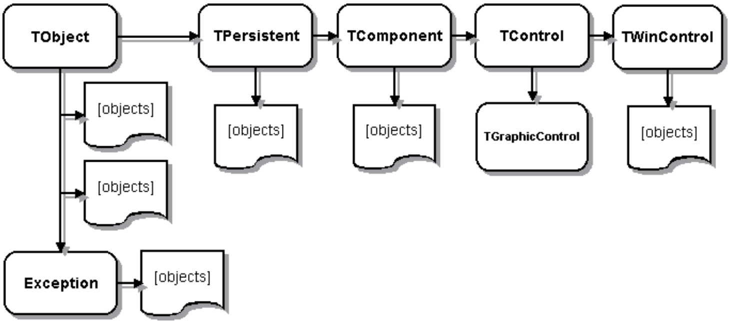
Figure 3.2 Objects, components, and controls



Every object inherits from *TObject*, and many objects inherit from *TComponent*. Controls, which inherit from *TControl*, have the ability to display themselves at runtime. A control like *TCheckBox* inherits all the functionality of *TObject*, *TComponent*, and *TControl*, and adds specialized capabilities of its own.

Figure 3.3 is an overview of the Visual Component Library (VCL) that shows the major branches of the inheritance tree. The Borland Component Library for Cross-Platform (CLX) look very much the same at this level but *TWinControl* is replaced by *TWidgetControl*.

Figure 3.3 A simplified hierarchy diagram



Several important base classes are shown in the figure, and they are described in the following table:

Table 3.1 Important base classes

Class	Description
TObject	Signifies the base class and ultimate ancestor of everything in the VCL or CLX. *TObject* encapsulates the fundamental behavior common to all VCL/CLX objects by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.
Exception	Specifies the base class of all classes that relate to exceptions. *Exception* provides a consistent interface for error conditions, and enables applications to handle error conditions gracefully.
TPersistent	Specifies the base class for all objects that implement properties. Classes under *TPersistent* deal with sending data to streams and allow for the assignment of classes.
TComponent	Specifies the base class for all nonvisual components such as *TApplication*. *TComponent* is the common ancestor of all components. This class allows a component to be displayed on the Component palette, lets the component own other components, and allows the component to be manipulated directly on a form.
TControl	Represents the base class for all controls that are visible at runtime. *TControl* is the common ancestor of all visual components and provides standard visual controls like position and cursor. This class also provides events that respond to mouse actions.
TWinControl	Specifies the base class of all user interface objects also called widgets. Controls under *TWinControl* are windowed controls that can capture keyboard input. (In CLX, *TWidgetControl* replaces *TWinControl*.)

The next few sections present a general description of the types of classes that each branch contains. For a complete overview of the VCL object hierarchy, refer to the VCL Object Hierarchy wall chart that is included with this product. For details on CLX, refer to the CLX Object Hierarchy wall chart included with the product and the Kylix documentation.

TObject branch

The *TObject* branch includes all objects that descend from *TObject* but not from *TPersistent*. All VCL or CLX objects descend from *TObject*, an abstract class whose methods define fundamental behavior like construction, destruction, and message or system event handling. Much of the powerful capability of VCL and CLX objects are established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all objects in the VCL and CLX by introducing methods that provide:

- The ability to respond when object instances are created or destroyed.
- Class type and instance information on an object, and runtime type information (RTTI) about its published properties.
- Support for message-handling (VCL only).

TObject is the immediate ancestor of many simple classes. Classes that are contained within this branch have one common, important characteristic: they are transitory. What this means is that these classes do not have a method to save the state that they are in prior to destruction; they are not persistent.

One of the main groups of classes in this branch is the *Exception* class. This class provides a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions.

Another type of group in the *TObject* branch are classes that encapsulate data structures, such as:

- *TBits,* a class that stores an "array" of Boolean values
- *TList,* a linked list class
- *TStack,* a class that maintains a last-in first-out array of pointers
- *TQueue,* a class that maintains a first-in first-out array of pointers

In the VCL, you can also find wrappers for external objects like *TPrinter*, which encapsulates the Windows printer interface, and *TRegistry,* a low-level wrapper for the system registry and functions that operate on the registry. These are specific to the Windows environment.

TStream is good example of another type of class in this branch. *TStream* is the base class type for stream objects that can read from or write to various kinds of storage media, such as disk files, dynamic memory, and so on.

So you can see, this branch includes many different types of classes that are very useful to you as a developer.

TPersistent branch

Objects in this branch of the VCL and CLX descend from *TPersistent* but not from *TComponent*. *TPersistent* adds persistence to objects. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory.

Objects in this branch implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. This branch introduces the *GetOwner* function which lets you determine the owner of the property.

Objects in this branch are also the first to include a published section where properties can be automatically loaded and saved. A *DefineProperties* method also allows you to indicate how to load and save properties.

Following are some of the other classes in the TPersistent branch of the hierarchy:

- *TGraphicsObject,* an abstract base class for graphics objects such as: *TBrush*, *TFont*, and *TPen*.
- *TGraphic,* an abstract base class for objects such as icons and bitmaps that can store and display visual images: *TBitmap* and *TIcon* (and for Windows development only: *TMetafile).*
- *TStrings,* a base class for objects that represent a list of strings.
- *TClipboard,* a class that contains text or graphics that have been cut or copied from an application.
- *TCollection*, *TOwnedCollection*, and *TCollectionItem,* classes that maintain indexed collections of specially defined items.

TComponent branch

TComponent branch contains objects that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time. They are persistent objects that can do the following:

- Appear on the Component palette and can be changed in the form designer.
- Own and manage other components.
- Load and save themselves.

Several methods in *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming is introduced in this branch of the VCL and CLX. Delphi handles most streaming chores automatically. Properties are persistent if they are published and published properties are automatically streamed.

The *TComponent* class also introduces the concept of ownership that is propagated throughout the VCL and CLX. Two properties support ownership: *Owner* and *Components*. Every component has an *Owner* property that references another component as its owner. A component may own other components. In this case, all owned components are referenced in the component's *Array* property.

A component's constructor takes a single parameter that is used to specify the new component's owner. If the passed-in owner exists, the new component is added to the owner's Components list. Aside from using the Components list to reference owned components, this property also provides for the automatic destruction of owned components. As long as the component has an owner, it will be destroyed when the owner is destroyed. For example, since *TForm* is a descendant of *TComponent*, all components owned by the form are destroyed and their memory freed when the form is destroyed. This assumes that all of the components on the form clean themselves up properly when their destructors are called.

If a property type is a *TComponent* or a descendant, the streaming system creates an instance of that type when reading it in. If a property type is *TPersistent* but not *TComponent*, the streaming system uses the existing instance available through the property and read values for that instance's properties.

When creating a form file (a file used to store information about the components on the form), the form designer loops through its components array and saves all the components on the form. Each component "knows" how to write its changed properties out to a stream (in this case, a text file). Conversely, when loading the properties of components in the form file, the form designer loops through the components array and loads each component.

The types of classes you'll find in this branch include:

- *TMainMenu,* a class that provides a menu bar and its accompanying drop-down menus for a form.
- *TTimer,* a class that includes the timer functions.
- *TOpenDialog*, *TSaveDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog*, and so on, provide commonly used dialog boxes.
- *TActionList,* a class that maintains a list of actions used with components and controls, such as menu items and buttons.
- *TScreen,* a class that keeps track of what forms and data modules have been instantiated by the application, the active form, and the active control within that form, the size and resolution of the screen, and the cursors and fonts available for the application to use.

Components that do not need a visual interface can be derived directly from *TComponent*. To make a tool such as a *TTimer* device, you can derive from *TComponent*. This type of component resides on the Component palette but performs internal functions that are accessed through code rather than appearing in the user interface at runtime.

In CLX, the *TComponent* branch also includes *THandleComponent*. This is the base class for nonvisual components that require a handle to an underlying Qt object such as dialogs and menus.

TControl branch

The *TControl* branch consists of components that descend from *TControl* but not *TWinControl* (*TWidgetControl* in CLX). Objects in this branch are controls that are visual objects which the application user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position, the cursor associated with the control's window (or widget in CLX), methods to paint or move the control, and events to respond to mouse actions. Controls can never receive keyboard input.

Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls. This includes drawing routines, standard events, and containership.

There are two basic types of control:

- Those that have a window (or widget) of their own
- Those that use the window (or widget) of their "parent"

Controls that have their own window are called "windowed" controls (VCL) or "widget-based" controls (CLX) and descend from *TWinControl* (*TWidgetControl* in CLX). Buttons and check boxes fall into this class.

Controls that use a parent window (or widget) are called "graphic" controls and descend from *TGraphicControl*. Image and label controls fall into this class. In the VCL, the main difference between these types of components is that graphic controls do not maintain a window handle, and thus cannot receive the input focus. In CLX, the main difference between these types of components is that graphic controls do not have an associated widget, and thus cannot receive the input focus nor can they contain other controls. Because a graphic control does not need a handle, its demand on system resources is lessened, and painting a graphic control is quicker than painting a widget-based control.

TGraphicControl controls must draw themselves and include controls such as:

Table 3.2 Graphic controls

Control	Description
TImage	Displays graphical images.
TLabel	Displays text on a form.
TBevel	Represents a beveled outline.
TPaintBox	Provides a canvas that applications can use for drawing or rendering an image.

Notice that these include common paint routines (*Repaint*, *Invalidate*, and so on) that never need to receive focus.

TWinControl/TWidgetControl branch

The *TWinControl* branch(*TWidgetControl* replaces *TWinControl* in CLX) includes all controls that descend from *TWinControl*. *TWinControl* is the base class for all windowed controls, including many of the items that you will use in the user interface of an application.

TWidgetControl is the base class for all widget-based controls or *widgets*. The term widget comes from combining "window" and "gadget." A widget is almost anything you use in the user interface of an application. Examples of widgets are buttons, labels, and scroll bars.

The following are features of windowed and widget-based controls:

- Both can receive focus while an application is running.
- Other controls may display data, but the user can use the keyboard to interact with windowed or widget-based controls.
- Windowed or widget-based controls can contain other controls.

- A control that contains other controls is called a parent. Only a windowed or widget-based control can be a parent of one or more child controls.
- Windowed controls have a window handle. Widget-based controls have an associated widget.

Descendants of *TWinControl* (*TWidgetControl* in CLX) are controls that can receive focus, meaning they can receive keyboard input from the application user. This implies that many more standard events apply to them.

This branch includes both controls that are drawn automatically (including *TEdit*, *TListBox*, *TComboBox*, *TPageControl*, and so on) and custom controls that Delphi must draw (such as *TDBNavigator*, *TMediaPlayer* (VCL only), *TGauge* (VCL only), and so on). Direct descendants of *TWinControl* (*TWidgetControl* in CLX) typically implement standard controls, like an edit field, a combo box, list box, or page control, and, therefore, already know how to paint themselves.

The *TCustomControl* class is provided for components that require a window handle but do not encapsulate a standard control that includes the ability to repaint itself. You never have to worry about how the controls render themselves or how they respond to events—Delphi completely encapsulates this behavior for you.

The following sections provide an overview of controls. Refer to Chapter 7, “Working with controls” for more information on using controls.

Properties common to TControl

All visual controls (descendants of *TControl*) share certain properties including:

- Action properties
- Position, size, and alignment properties
- Display properties
- Parent properties
- A navigation property
- Drag-and-drop properties
- Drag-and-dock properties (VCL only)

While these properties are inherited from *TControl*, they are published—and hence appear in the Object Inspector—only for components to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays.

Action properties

Actions let you share common code for performing actions (for example, when a tool bar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

- *Action* designates the action associated with the control.
- *ActionLink* contains the action link object associated with the control.

Position, size, and alignment properties

This set of properties defines the position and size of a control on the parent control:

- *Height* sets the vertical size.
- *Width* sets the horizontal size.
- *Top* positions the top edge.
- *Left* positions the left edge.
- *AutoSize* specifies whether the control sizes itself automatically to accommodate its contents.
- *Align* determines how the control aligns within its container (parent control).
- *Anchor* specifies how the control is anchored to its parent (VCL only).

This set of properties determine the height, width, and overall size of the control's client area:

- *ClientHeight* specifies the height of the control's client area in pixels.
- *ClientWidth* specifies the width of the control's client area in pixels.

These properties aren't accessible in nonvisual components, but Delphi does keep track of where you place the component icons on your forms. Most of the time you'll set and alter these properties by manipulating the control's image on the form or using the Alignment palette. You can, however, alter them at runtime.

Display properties

The following properties govern the general appearance of a control:

- *Color* changes the background color of a control.
- *Font* changes the color, type family, style, or size of text.
- *Cursor* specifies the image used to represent the mouse pointer when it passes into the region covered by the control.
- *DesktopFont* specifies whether the control uses the Windows icon font when writing text (VCL only).

Parent properties

To maintain a consistent appearance across your application, you can make any control look like its container—called its *parent*—by setting the parent properties to *True*.

- *ParentColor* determines where a control looks for its color information.
- *ParentFont* determines where a control looks for its font information.
- *ParentShowHint* determines where a control looks to find out if its Help Hint should be shown.

A navigation property

The following property determines how users navigate among the controls in a form:

- *Caption* contains the text string that labels a component. To underline a character in a string, include an ampersand (&) before the character. This type of character is called an accelerator key. The user can then select the control or menu item by pressing *Alt* while typing the underlined character.

Drag-and-drop properties

Two component properties affect drag-and-drop behavior:

- *DragMode* determines how dragging starts. By default, *DragMode* is *dmManual*, and the application must call the *BeginDrag* method to start dragging. When *DragMode* is *dmAutomatic*, dragging starts as soon as the mouse button goes down.
- *DragCursor* determines the shape of the mouse pointer when it is over a draggable component (VCL only).

Drag-and-dock properties (VCL only)

The following properties control drag-and-dock behavior:

- *Floating* indicates whether the control is floating.
- *DragKind* specifies whether the control is being dragged normally or for docking.
- *DragMode* determines how the control initiates drag-and-drop or drag-and-dock operations.
- *FloatingDockSiteClass* specifies the class of the temporary control that hosts the control when it is floating.
- *DragCursor* is the cursor that is shown while dragging.
- *DockOrientation* specifies how the control is docked relative to other controls docked in the same parent.
- *HostDockSite* specifies the control in which the control is docked.

For more information, see “Implementing drag-and-dock in controls” on page 7-4.

Standard events common to TControl

The VCL defines a set of standard events for its controls. The following events are declared as part of the *TControl* class, and are therefore available for all classes derived from *TControl*:

- *OnClick* occurs when the user clicks the control.
- *OnContextPopup* occurs when the user right-clicks the control or otherwise invokes the popup menu (such as using the keyboard).
- *OnCanResize* occurs when an attempt is made to resize the control.
- *OnResize* occurs immediately after the control is resized.
- *OnConstrainedResize* occurs immediately after OnCanResize.
- *OnStartDock* occurs when the user begins to drag a control with a DragKind of dkDock (VCL only).
- *OnEndDock* occurs when the dragging of an object ends, either by docking the object or by canceling the dragging (VCL only).
- *OnStartDrag* occurs when the user begins to drag the control or an object it contains by left-clicking on the control and holding the mouse button down.

- *OnEndDrag* occurs when the dragging of an object ends, either by dropping the object or by canceling the dragging.
- *OnDragDrop* occurs when the user drops an object being dragged.
- *OnMouseMove* occurs when the user moves the mouse pointer while the mouse pointer is over a control.
- *OnDblClick* occurs when the user double-clicks the primary mouse button when the mouse pointer is over the control.
- *OnDragOver* occurs when the user drags an object over a control (VCL only).
- *OnMouseDown* occurs when the user presses a mouse button with the mouse pointer over a control.
- *OnMouseUp* occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

Properties common to TWinControl and TWidgetControl

All windowed controls (descendants of *TWinControl* in the VCL and *TWidgetControl* in CLX) share certain properties including:

- Information about the control
- Border style display properties
- Navigation properties
- Drag-and-dock properties (VCL only)

While these properties are inherited from *TWinControl* and *TWidgetControl*, they are published—and hence appear in the Object Inspector—only for controls to which they are applicable.

General information properties

The general information properties contain information about the appearance of the *TWinControl* and *TWidgetControl*, client area size and origin, windows assigned information, and help context information.

- *ClientOrigin* specifies the screen coordinates (in pixels) of the top left corner of a control’s client area. The screen coordinates of a control that is descended from *TControl* and not *TWinControl* are the screen coordinates of the control’s parent added to its *Left* and *Top* properties.
- *ClientRect* returns a rectangle with its *Top* and *Left* properties set to zero, and its *Bottom* and *Right* properties set to the control's *Height* and *Width*, respectively. *ClientRect* is equivalent to Rect(0, 0, *ClientWidth*, *ClientHeight*).
- *Brush* determines the color and pattern used for painting the background of the control.
- *HelpContext* provides a context number for use in calling context-sensitive online Help.
- *Handle* provides access to the window or widget handle of the control.

Border style display properties

The bevel properties control the appearance of the beveled lines, boxes, or frames on the forms and windowed controls in your application.

Many more objects in the VCL publish these properties; they are not all available in CLX and the border style properties are published on fewer objects.

- *InnerBevel* specifies whether the inner bevel has a raised, lowered, or flat look (VCL only).
- *BevelKind* specifies the type of bevel if the control has beveled edges (VCL only).
- *BevelOuter* specifies whether the outer bevel has a raised, lowered, or flat look.
- *BevelWidth* specifies the width, in pixels, of the inner and outer bevels.
- *BorderWidth* is used to get or set the width of the control's border.
- *BevelEdges* is used to get or set which edges of the control are beveled.

Navigation properties

Two additional properties determine how users navigate among the controls on a form:

- *TabOrder* indicates the position of the control in its parent's tab order, the order in which controls receive focus when the user presses the *Tab* key. Initially, tab order is the order in which the components are added to the form, but you can change this by changing *TabOrder*. *TabOrder* is meaningful only if *TabStop* is *True*.
- *TabStop* determines whether the user can tab to a control. If *TabStop* is *True*, the control is in the tab order.

Drag-and-dock properties (VCL only)

The following properties manage drag-and-dock behavior in VCL objects:

- *UseDockManager* specifies whether the dock manager is used in drag-and-dock operations.
- *VisibleDockClientCount* specifies the number of visible controls that are docked on the windowed control.
- *DockManager* specifies the control's dock manager interface.
- *DockClients* lists the controls that are docked to the windowed control.
- *DockSite* specifies whether the control can be the target of drag-and-dock operations.

For more information, see "Implementing drag-and-dock in controls" on page 7-4.

Events common to TWinControl and TWidgetControl

The following events exist for all controls derived from *TWinControl* in the VCL (this also includes all the controls that Windows defines) and in *TWidgetControl* in CLX. These events are in addition to those that exist in all controls.

- *OnEnter* occurs when the control is about to receive focus.
- *OnKeyDown* occurs on the down stroke of a key press.
- *OnKeyPress* occurs when a user presses a single character key.
- *OnKeyUp* occurs when the user releases a key that has been pressed.

- *OnExit* occurs when the input focus shifts away from one control to another.
- *OnMouseWheel* occurs when the mouse wheel is rotated.
- *OnMouseWheelDown* occurs when the mouse wheel is rotated downward.
- *OnMouseWheelUp* occurs when the mouse wheel is rotated upward.

The following events relate to docking and are available in the VCL only:

- *OnUnDock* occurs when the application tries to undock a control that is docked to a windowed control (VCL only).
- *OnDockDrop* occurs when another control is docked to the control (VCL only).
- *OnDockOver* occurs when another control is dragged over the control (VCL only).
- *OnGetSiteInfo* returns the control's docking information (VCL only).

Creating the application user interface

All visual design work in Delphi takes place on forms. When you open Delphi or create a new project, a blank form is displayed on the screen. You can use it to start building your application interface including windows, menus, and common dialogs.

You design the look and feel of the graphical user interface for an application by placing and arranging visual components such as buttons and list boxes on the form. Delphi takes care of the underlying programming details. You can also place invisible components on forms to capture information from databases, perform calculations, and manage other interactions.

Chapter 6, "Developing the application user interface" provides details on using forms such as creating modal forms dynamically, passing parameters to forms, and retrieving data from forms.

Using Delphi components

Many visual components are provided in the development environment itself on the Component palette. All visual design work in Delphi takes place on forms. When you open Kylix or create a new project, a blank form is displayed on the screen. You select components from the Component palette and drop them onto the form. You design the look and feel of the application's user interface by arranging the visual components such as buttons and list boxes on the form. Once a visual component is on the form, you can adjust its position, size, and other design-time properties. Delphi takes care of the underlying programming details.

Delphi components are grouped functionally on different pages of the Component palette. For example, commonly used components such as those to create menus, edit boxes, or buttons are located on the Standard page of the Component palette. Handy VCL controls such as a timer, paint box, media player, and OLE container are on the System page.

At first glance, Delphi's components appear to be just like any other classes. But there are differences between components in Delphi and the standard class hierarchies that many programmers work with. Some differences are described here:

- All Delphi components descend from *TComponent*.
- Components are most often used as is and are changed through their properties, rather than serving as "base classes" to be subclassed to add or change functionality. When a component is inherited, it is usually to add specific code to existing event handling member functions.
- Components can only be allocated on the heap, not on the stack.
- Properties of components intrinsically contain runtime type information.
- Components can be added to the Component palette in the Delphi user interface and manipulated on a form.

Components often achieve a better degree of encapsulation than is usually found in standard classes. For example, consider the use of a dialog containing a push button. In a Windows program developed using VCL components, when a user clicks on the button, the system generates a WM_LBUTTONDOWN message. The program must catch this message (typically in a **switch** statement, a message map, or a response table) and dispatch it to a routine that will execute in response to the message.

Most Windows messages (VCL) or system events (CLX) are handled by Delphi components. When you want to respond to a message, you only need to provide an event handler.

Setting component properties

Published properties can be set at design time in the Object Inspector and, in some cases, with special property editors.

To set properties at runtime, assign them new values in your application source code.

For information about the properties of each component, see the online Help.

Using the Object Inspector

When you select a component on a form, the Object Inspector displays its published properties and (when appropriate) allows you to edit them. Use the *Tab* key to toggle between the Value column and the Property column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column.

If a plus (+) symbol appears next to a property name, clicking the plus symbol or typing '+' when the property has focus displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol or typing '-' hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the Object Inspector and choose View. For more information, see “property categories” in the online Help.

When more than one component is selected, the Object Inspector displays all properties—except *Name*—that are shared by the selected components. If the value for a shared property differs among the selected components, the Object Inspector displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

Using property editors

Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the Object Inspector. To open the property editor, double-click in the Value column, click the ellipsis mark, or type *Ctrl+Enter* when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

Setting properties at runtime

Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

```
Form1.Caption := MyString;
```

Calling methods

Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control’s image on the screen. You could call the *Repaint* method in a draw-grid object like this:

```
DrawGrid1.Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for example, to repaint a form within an event handler of one of the form’s child controls, you don’t have to prepend the name of the form to the method call:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
```

For more information about scope, see “Scope and qualifiers” on page 3-8.

Working with events and event handlers

In Delphi, almost all the code you write is executed, directly or indirectly, in response to *events*. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is an Object Pascal procedure. The sections that follow show how to

- Generate a new event handler
- Generate a handler for a component’s default event
- Locate event handlers
- Associate an event with an existing event handler
- Associate menu events with event handlers
- Delete event handlers

Generating a new event handler

Delphi can generate skeleton event handlers for forms and other components. To create an event handler,

1 Select a component.

2 Click the Events tab in the Object Inspector. The Events page of the Object Inspector displays all events defined for the component.

3 Select the event you want, then double-click the Value column or press *Ctrl+Enter*. Delphi generates the event handler in the Code editor and places the cursor inside the **begin...end** block.

4 Inside the **begin...end** block, type the code that you want to execute when the event occurs.

Generating a handler for a component’s default event

Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button’s default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as *TBevel*, don’t respond to any events. Other components respond differently when you double-click on them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

Locating event handlers

If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that’s not the default,

1 In the form, select the component whose event handler you want to locate.

2 In the Object Inspector, click the Events tab.

3 Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor at the beginning of the event-handler body.

Associating an event with an existing event handler

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler,

1. On the form, select the component whose event you want to handle.
2. On the Events page of the Object Inspector, select the event to which you want to attach a handler.
3. Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The procedure above is an easy way to reuse event handlers. *Action lists* and in the VCL, *action bands*, however, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications, whereas action bands cannot. For more information about action lists and action bands, see "Organizing actions for toolbars and menus" on page 6-16.

Using the Sender parameter

In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
else
  AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

Displaying and coding shared events

When components share events, you can display their shared events in the Object Inspector. First, select the components by holding down the *Shift* key and clicking on them in the Form Designer; then choose the Events tab in the Object Inspector. From the Value column in the Object Inspector, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.

Associating menu events with event handlers

The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This section explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see "Creating and managing menus" on page 6-29.

To create an event handler for a menu item,

1. Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
2. Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
3. From the Menu Designer, double-click the menu item. Delphi generates an event handler in the Code editor and places the cursor inside the **begin...end** block.
4. Inside the **begin...end** block, type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing *OnClick* event handler,

1. Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
2. Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
3. On the Events page of the Object Inspector, click the down arrow in the Value column next to *OnClick* to open a list of previously written event handlers. (The list includes only event handlers written for *OnClick* events on this form.) Select from the list by clicking an event handler name.

Deleting event handlers

When you delete a component from a form using the Form Designer, Delphi removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (in the **interface** section of the unit) and its implementation (in the **implementation** section); otherwise you'll get a compiler error when you build your project.

VCL and CLX components

The Component palette contains a selection of components that handle a wide variety of programming tasks. You can add, remove, and rearrange components on the palette, and you can create component *templates* and *frames* that group several components.

The components on the palette are arranged in pages according to their purpose and functionality. Which pages appear in the default configuration depends on the version of Delphi you are running. Table 3.3 lists typical default pages and components available for creating applications. Some of the tabs and components are not cross platform and the table points them out. You can use some VCL-specific nonvisual components in Windows-only CLX applications, however, the applications will not be cross-platform unless you isolate these portions of the code.

Table 3.3 Component palette pages

Page name	Description	Cross platform?
Standard	Standard controls, menus	Yes
Additional	Specialized controls	Yes except ApplicationEvents and CustomizeDlg
Win32	Windows common controls	Many of the same components are on the Common Controls tab that appears instead when creating CLX applications; RichEdit, UpDown, HotKey, Animate, DataTimePicker, MonthCalendar, Coolbar, PageScroller, and ComboBoxEx are not cross-platform
System	Components and controls for system-level access, including timers, multimedia, and DDE	Timer is but PaintBox, MediaPlayer, OleContainer, and the Dde components are not
Data Access	Components for working with database data that are not tied to any particular data access mechanism	Yes
Data Controls	Visual, data-aware controls	Yes except for DBRichEdit, DBCtrlGrid, and DBChart
dbExpress	Database controls that use dbExpress, a cross-platform, database-independent layer that provides methods for dynamic SQL processing. It defines a common interface for accessing SQL servers.	Yes
DataSnap	Components used for creating multi-tiered database applications	No but can be used in Windows CLX applications
BDE	Components that provide data access through the Borland Database Engine	No but can be used in Windows CLX applications
ADO	Components that provide data access through the ADO framework	No but can be used in Windows CLX applications
InterBase	Components that provide direct access to InterBase	Yes
InternetExpress	Components that are simultaneously a Web Server application and the client of a multi-tiered database application	No but can be used in Windows CLX applications
Internet	Components for Internet communication protocols and Web applications	Yes except for ClientSocket, ServerSocket, QueryTableProducer, XMLDoc, and WebBrowser
WebSnap	Components for building Web server applications	No but can be used in Windows CLX applications
FastNet	NetMasters Internet controls	No but can be used in Windows CLX applications

Table 3.3 Component palette pages (continued)

Page name	Description	Cross platform?
QReport	QuickReport components for creating embedded reports	No but can be used in Windows CLX applications
Dialogs	Commonly used dialog boxes	Yes except for OpenPictureDialog, SavePictureDialog, PrinterSetup-Dialog, and PageSetupDialog
Win 3.1	Old style Win 3.1 components	No
Samples	Sample custom components	No
ActiveX	Sample ActiveX controls; see Microsoft documentation (msdn.microsoft.com)	No
COM+	Component for handling COM+ events	No but can be used in Windows CLX applications
WebServices	Components for writing applications that implement or use SOAP-based Web services	No but can be used in Windows CLX applications
Servers	COM Server examples for Microsoft Excel, Word, and so on (see Microsoft MSDN documentation)	No but can be used in Windows CLX applications
Indy Clients	Cross-platform Internet components for the client (open source Winshoes Internet components)	Yes
Indy Servers	Cross-platform Internet components for the server (open source Winshoes Internet components)	Yes
Indy Misc	Additional cross-platform Internet components (open source Winshoes Internet components)	Yes

The online Help provides information about the components on the Component palette. Some of the components on the ActiveX, Servers, and Samples pages, however, are provided as examples only and are not documented.

Adding custom components to the Component palette

You can install custom components—written by yourself or third parties—on the Component palette and use them in your applications. To write a component, see Part V, “Creating custom components”. To install an existing component, see “Installing component packages” on page 11-5.

Text controls

Many applications present text to the user or allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

Use this component:	When you want users to do this:
TEdit	Edit a single line of text
TMemo	Edit multiple lines of text
TMaskEdit	Adhere to a particular format, such as a postal code or phone number
TRichEdit	Edit multiple lines of text using rich text format (VCL only)

TEdit and *TMaskEdit* are simple text controls that include a single line text edit box in which you can type information. When the edit box has focus, a blinking insertion point appears.

You can include text in the edit box by assigning a string value to its *Text* property. You control the appearance of the text in the edit box by assigning values to its *Font* property. You can specify the typeface, size, color, and attributes of the font. The attributes affect all of the text in the edit box and cannot be applied to individual characters.

An edit box can be designed to change its size depending on the size of the font it contains. You do this by setting the *AutoSize* property to True. You can limit the number of characters an edit box can contain by assigning a value to the *MaxLength* property.

TMaskEdit is a special edit control that validates the text entered against a mask that encodes the valid forms the text can take. The mask can also format the text that is displayed to the user.

TMemo is for adding several lines of text.

Text control properties

Following are some of the important properties of text controls:

Table 3.4 Text control properties

Property	Description
Text	Determines the text that appears in the edit box or memo control.
Font	Controls the attributes of text written in the edit box or memo control.
AutoSize	Enables the edit box to dynamically change its height depending on the currently selected font.
ReadOnly	Specifies whether the user is allowed to change the text.
MaxLength	Limits the number of characters in simple text controls.

Properties of memo and rich text controls

Memo and rich text controls, which handle multiple lines of text, have several properties in common. Note that rich text controls are not cross-platform.

TMemo is another type of edit box, which handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property.

Memo and rich text controls include other properties such as the following:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the *Modified* property.
- *Lines* contains the text as a list of strings.
- *OEMConvert* determines whether the text is temporarily converted from ANSI to OEM as it is entered. This is useful for validating file names (VCL only).
- *WordWrap* determines whether the text will wrap at the right margin.
- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.
- *SelText* contains the currently selected (highlighted) part of the text.
- *SelStart* and *SelLength* indicate the position and length of the selected part of the text.

At runtime, you can select all the text in the memo with the *SelectAll* method.

Rich text controls (VCL only)

The rich edit (*TRichEdit*) component is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering.

Specialized input controls

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
TScrollBar	Select values on a continuous range
TTrackBar	Select values on a continuous range (more visually effective than a scroll bar)
TUpDown	Select a value from a spinner attached to an edit component (VCL only)
THotKey	Enter *Ctrl*/*Shift*/*Alt* keyboard sequences (VCL only)
TSpinEdit	Select a value from a spinner widget (CLX only)

Scroll bars

The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

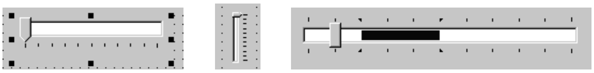
The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don’t require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.

Track bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

- Use the *Max* and *Min* properties to set the upper and lower range of the track bar.
- Use *SelEnd* and *SelStart* to highlight a selection range. See Figure 3.4.
- The *Orientation* property determines whether the track bar is vertical or horizontal.
- By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTick* method.

Figure 3.4 Three views of the track bar component



- *Position* sets a default position for the track bar and tracks the position at runtime.
- By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.
- Set *PageSize* to determine the number of ticks moved when the user presses *Page Up* and *Page Down*.

Up-down controls (VCL only)

An up-down control (*TUpDown*) consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.

Spin edit controls (CLX only)

A spin edit control (*TSpinEdit*) is also called an up-down widget, little arrows widget, or spin button. This control lets the application user change an integer value in fixed increments, either by clicking the up or down arrow buttons to increase or decrease the value currently displayed, or by typing the value directly into the spin box.

The current value is given by the *Value* property; the increment, which defaults to 1, is specified by the *Increment* property.

Hot key controls (VCL only)

Use the hot key component (*THotKey*) to assign a keyboard shortcut that transfers focus to any control. The *HotKey* property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*.

The hot key component can be assigned as the *ShortCut* property of a menu item. Then, when a user enters the key combination specified by the *HotKey* and *Modifiers* properties, Windows activates the menu item.

Splitter controls

A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment* to *alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to *True* to give the splitter's edge a 3D look.

Buttons and similar controls

Aside from menus, buttons provide the most common way to invoke a command in an application. Delphi offers several button-like controls:

Use this component:	To do this:
TButton	Present command choices on buttons with text
TBitBtn	Present command choices on buttons with text and glyphs
TSpeedButton	Create grouped toolbar buttons
TCheckBox	Present on/off options
TRadioButton	Present a set of mutually exclusive choices
TToolBar	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions
TCoolBar	Display a collection of windowed controls within movable, resizable bands (VCL only)

Button controls

Users click button controls with the mouse to initiate actions. Buttons are labeled with text that represent the action. The text is specified by assigning a string value to the *Caption* property. Most buttons can also be selected by pressing a key on the keyboard as a keyboard shortcut. The shortcut is shown as an underlined letter on the button.

Users click button controls to initiate actions. You can assign an action to a *TButto*n component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set *Cancel* to *True* if you want the button to trigger its *OnClick* event when the user presses Esc.
- Set *Default* to *True* if you want the Enter key to trigger the button's *OnClick* event.

Bitmap buttons

A bitmap button (*BitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph is to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered in the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

Speed buttons

Speed buttons, which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

- To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.
- By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*.
- If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio group.

For more information on speed buttons, refer to subtopics in the section “Adding a toolbar using a panel component” on page 6-43.

Check boxes

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*.

- Set *Checked* to *True* to make the box appear checked by default.
- Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed.
- The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

Note Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

Radio buttons

Radio buttons present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See “Grouping components” on page 3-39 for more information.

A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value True or False to the Checked property to change the radio button's visual state.

Toolbars

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *ToolBar* component, then right-click and choose New Button to add buttons to the toolbar.

The *TToolBar* component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and *TToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

You can use a centralized set of actions on toolbars and menus, by using *action lists* or *action bands*. See "Using action lists" on page 6-23 for details on how to use action lists with buttons and toolbars.

Toolbars can also parent other controls such as edit boxes, combo boxes, and so on.

Cool bars (VCL only)

A cool bar contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band.

The cool bar requires version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows\System32 directory) at both design time and runtime. Cool bars cannot be used in cross-platform applications.

- The *Bands* property holds a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the *Bands* property in the Object Inspector, then double-click in the Value column to the right, or click the ellipsis (...) button. You can also create bands by adding new windowed controls from the palette.
- The *FixedOrder* property determines whether users can reorder the bands.
- The *FixedSize* property determines whether the bands maintain a uniform height.

Handling lists

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
TListBox	A list of text strings
TCheckListBox	A list with a check box in front of each item
TComboBox	An edit box with a scrollable drop-down list
TTreeView	A hierarchical list

Use this component:	To display:
TListView	A list of (draggable) items with optional icons, columns, and headings
TDateTimePicker	A list box for entering dates or times (VCL only)
TMonthCalendar	A calendar for selecting dates (VCL only)

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see "Working with string lists" on page 3-47.

List boxes and check-list boxes

List boxes (*TListBox*) and check-list boxes display lists from which users can select items.

- *Items* uses a *TStrings* object to fill the control with values.
- *ItemIndex* indicates which item in the list is selected.
- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *IntegralHeight* specifies whether the list box shows only entries that fit completely in the vertical space (VCL only).
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see "Adding graphics to controls" on page 7-11.

To create a simple list box,

1. Within your project, drop a list box component from the Component palette onto a form.
2. Size the list box and set its alignment as needed.
3. Double-click the right side of the *Items* property or choose the ellipsis button to display the String List Editor.
4. Use the editor to enter free form text arranged in lines for the contents of the list box.
5. Then choose OK.

To let users select multiple items in the list box, you can use the *ExtendedSelect* and *MultiSelect* properties.

Combo boxes

A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data.

Three types of combo boxes are: standard, drop-down (the default), and drop-down list.

- Use the *Style* property to select the type of combo box you need.
- Use *csDropDown* if you want an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list). Set the *DropDownCount* property to change the number of items displayed in the list.
- Use *csSimple* to create a combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are displayed.
- Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see “Adding graphics to controls” on page 7-11.

At runtime, CLX combo boxes work differently than VCL combo boxes. In CLX (but not in the VCL combo box), you can add a item to a drop down by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

Tree views

A tree view *(TTreeView)* displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items’ text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of '+' and '–' buttons to indicate whether an item can be expanded.
- *ShowLines* enables display of connecting lines to show hierarchical relationships (VCL only).
- *ShowRoot* determines whether lines connecting the top-level items are displayed (VCL only).

To add items to a tree view control at design time, double-click on the control to display the TreeView Items editor. The items you add become the value of the *Items* property. You can change the items at runtime by using the methods of the *Items* property, which is an object of type *TTreeNodes*. *TTreeNodes* has methods for adding, deleting, and navigating the items in the tree view.

Tree views can display columns and subitems similar to list views in vsReport mode.

List views

List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window (VCL only).
- *vsList* displays items as labeled icons that cannot be dragged.

- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

Date-time pickers and month calendars (VCL only)

The DateTimePicker component displays a list box for entering dates or times, while the MonthCalendar component presents a calendar for entering dates or ranges of dates. To use these components, you must have version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows\System32 directory) at both design time and runtime. They are not available for use in cross-platform applications.

Grouping components

A graphical interface is easier to use when related controls and information are presented in groups. Delphi provides several components for grouping components:

Use this component:	When you want this:
TGroupBox	A standard group box with a title
TRadioGroup	A simple group of radio buttons
TPanel	A more visually flexible group of controls
TScrollBox	A scrollable region containing controls
TTabControl	A set of mutually exclusive notebook-style tabs
TPageControl	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls
THeaderControl	Resizable column headers

Group boxes and radio groups

A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the Component palette and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the Object Inspector; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

Panels

The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel alTop aligned so that its position will remain in place even if the form is resized.

The look of the panel can be changed to a raised or lowered look by using the *BevelOuter* and *BevelInner* properties. You can vary the values of these properties to create different visual 3-D effects. Note that if you merely want a raised or lowered bevel, you can use the less resource intensive *TBevel* control instead.

You can also use one or more panels to build various status bars or information display areas.

Scroll boxes

Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents.

Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

Another use of a scroll box is to restrict scrolling in areas of a window, such as a toolbar or status bar (*TPanel* components). To prevent a toolbar and status bar from scrolling, hide the scroll bars, and then position a scroll box in the client area of the window between the toolbar and status bar. The scroll bars associated with the scroll box will appear to belong to the window, but will scroll only the area inside the scroll box.

Tab controls

The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the Object Inspector; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

Page controls

The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control.

To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

To access the active page, use the *ActivePage* property. To change the active page, you can set either the *ActivePage* or the *ActivePageIndex* property.

Header controls

A header control (*THeaderControl*) is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).

Providing visual feedback

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime.

Use this component or property:	To do this:
TLabel and *TStaticText*	Display non-editable text
TStatusBar	Display a status region (usually at the bottom of a window)
TProgressBar	Show the amount of work completed for a particular task
Hint and *ShowHint*	Activate fly-by or "tooltip" help
HelpContext and *HelpFile*	Link context-sensitive online Help

Labels and static text components

Labels (*TLabel*) display text and are usually placed next to other controls. You place a label on a form when you need to identify or annotate another component such as an edit box or when you want to include text on a form. The standard label component, *TLabel*, is a non-windowed control (not widget-based in CLX), so it cannot receive focus; when you need a label with a window handle, use *TStaticText* instead.

Label properties include the following:

- *Caption* contains the text string for the label.
- *Font*, *Color*, and other properties determine the appearance of the label. Each label can use only one typeface, size, and color.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.

- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined and enables an accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

Labels usually display read-only static text that cannot be changed by the application user. The application can change the text while it is executing by assigning a new value to the *Caption* property. To add a text object to a form that a user can scroll or edit, use *TEdit*.

Status bars

Although you can use a panel to make a status bar, it is simpler to use the status bar component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

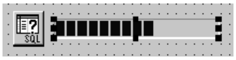
If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar.

You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the Object Inspector, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. Each panel's *Text* property contains the text displayed in the panel.

Progress bars

When your application performs a time-consuming operation, you can use a progress bar to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right.

Figure 3.5 A progress bar



The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

Help and hint properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.

Grids

Grids display information in rows and columns. If you're writing a database application, use the *TDBGrid* or *TDBCtrlGrid* component described in Chapter 15, "Using data controls." Otherwise, use a standard draw grid or string grid.

Draw grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

- The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.
- The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.
- You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.
- You can choose to have fixed or non-scrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.
- The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

String grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

Value list editors (VCL only)

TValueListEditor is a specialized grid for editing string lists that contain name/value pairs in the form Name=Value. The names and values are stored as a *TStrings* descendant that is the value of the *Strings* property. You can look up the value for any name using the *Values* property. *TValueListEditor* is not available for cross-platform programming.

The grid contains two columns, one for the names and one for the values. By default, the Name column is named "Key" and the Value column is named "Value". You can change these defaults by setting the *TitleCaptions* property. You can omit these titles using the *DisplayOptions* property (which also controls resize when you resize the control.)

You can control whether users can edit the Name column using the *KeyOptions* property. *KeyOptions* contains separate options to allow editing, adding new names, deleting names, and controlling whether new names must be unique.

You can control how users edit the entries in the Value column using the *ItemProps* property. Each item has a separate *TItemProp* object that lets you

- Supply an edit mask to limit the valid input.
- Specify a maximum length for values.
- Mark the value as read-only.
- Specify that the value list editor displays a drop-down arrow that opens a pick list of values from which the user can choose or an ellipsis button that triggers an event you can use for displaying a dialog in which users enter values.

  If you specify that there is a drop-down arrow, you must supply the list of values from which the user chooses. These can be a static list (the *PickList* property of the *TItemProp* object) or they can be dynamically added at runtime using the value list editor's *OnGetPickList* event. You can also combine these approaches and have a static list that the *OnGetPickList* event handler modifies.

  If you specify that there is an ellipsis button, you must supply the response that occurs when the user clicks that button (including the setting of a value, if appropriate). You provide this response by writing an *OnEditButtonClick* event handler.

Displaying graphics

The following components make it easy to incorporate graphics into an application.

Use this component:	To display:
TImage	Graphics files
TShape	Geometric shapes
TBevel	3-D lines and frames
TPaintBox	Graphics drawn by your program at runtime
TAnimate	AVI files (VCL only)

Images

The image component displays a graphical image, like a bitmap, icon, or metafile. The *Picture* property determines the graphic to be displayed. Use *Center*, *AutoSize*, *Stretch*, and *Transparent* to set display options. For more information, see "Overview of graphics programming" on page 8-1.

Shapes

The shape component displays a geometric shape. It is a nonwindowed control (not widget-based in CLX) and therefore, cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

Bevels

The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

Paint boxes

The paint box (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see"Overview of graphics programming" on page 8-1.

Animation control (VCL only)

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip. An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI files or AVI clips compressed using run-length encoding (RLE). Animation control cannot be used in cross-platform programming.

Following are some of the properties of an animation component:

- *ResHandle* is the Windows handle for the module that contains the AVI clip as a resource. Set *ResHandle* at runtime to the instance handle or module handle of the module that includes the animation resource. After setting *ResHandle*, set the *ResID* or *ResName* property to specify which resource in the indicated module is the AVI clip that should be displayed by the animation control.
- Set *AutoSize* to *True* to have the animation control adjust its size to the size of the frames in the AVI clip.
- *StartFrame* and *StopFrame* specify in which frames to start and stop the clip.
- Set *CommonAVI* to display one of the common Windows AVI clips provided in Shell32.DLL.
- Specify when to start and interrupt the animation by setting the *Active* property to *True* and *False*, respectively, and how many repetitions to play by setting the *Repetitions* property.
- The *Timers* property lets you display the frames using a timer. This is useful for synchronizing the animation sequence with other actions, such as playing a sound track.

Developing dialog boxes

The dialog box components on the Dialogs page of the Component palette make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening, saving, and printing files. Dialog boxes display and/or obtain data.

Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns *True*; if the user chooses Cancel to escape from the dialog box without making or saving changes, *Execute* returns *False*.

If you are developing cross-platform applications, you can use the dialogs provided with CLX in the QDialogs unit. For operating systems that have native dialog box types for common tasks, such as for opening or saving a file or for changing font or color, you can use the *UseNativeDialog* property. Set *UseNativeDialog* to *True* if your application will run in such an environment, and if you want it to use the native dialogs instead of the Qt dialogs.

Using open dialog boxes

One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files using a wildcard character and navigate through directories.

The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box.

When the user chooses OK in the dialog box, the user's file is stored in the *TOpenDialog FileName* property, which you can then process as you want.

The following code snippet can be placed in an *Action* and linked to the *Action* property of a *TMainMenu* subitem or be placed in the subitem's *OnClick* event:

```
if OpenDialog1.Execute then
  filename := OpenDialog1.FileName;
```

This code will show the dialog box and if the user presses the OK button, it will copy the name of the file into a previously declared *AnsiString* variable named `filename`.

Using helper objects

The VCL and CLX include a variety of nonvisual objects that simplify common programming tasks. This section describes a few Helper objects that make it easier to perform the following tasks:

- Working with lists
- Working with string lists
- Changing the Windows registry and .INI files
- Using streams

Working with lists

The following objects provide functionality for creating and managing lists:

Table 3.5 Components for creating and managing lists

Object	Maintains
TList	A list of pointers
TObjectList	A memory-managed list of instance objects
TComponentList	A memory-managed list of components (that is, instances of classes descended from *TComponent*)
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TClassList	A list of class types
TCollection, *TOwnedCollection*, and *TCollectionItem*	Indexed collections of specially defined items
TStringList	A list of strings

For more information about these objects, see the online reference.

Working with string lists

Applications often need to manage lists of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The VCL and CLX provide a common interface to any list of strings through an object called *TStrings* and its descendant *TStringList*. *TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to

- Sort the strings in the list.
- Prohibit duplicate strings in sorted lists.
- Respond to changes in the contents of the list.

In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are an instance of *TStrings*) and then use these lines as items in a combo box (also an instance of *TStrings*).

A string-list property appears in the Object Inspector with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists
- Creating a new string list
- Manipulating strings in a list
- Associating objects with a string list

Loading and saving string lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the WIN.INI file into a memo field and makes a backup copy called WIN.BAK.

```
procedure EditWinIni;
var
  FileName: string;{ storage for file name }
begin
  FileName := 'C:\WINDOWS\WIN.INI';{ set the file name }
  with Form1.Memo1.Lines do
  begin
    LoadFromFile(FileName);{ load from file }
    SaveToFile(ChangeFileExt(FileName, '.BAK'));{ save into backup file }
  end;
end;
```

Creating a new string list

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a **try...finally** block to ensure that the memory is freed even if an exception occurs.

1. Construct the string-list object.
2. In the **try** part of a **try...finally** block, use the string list.
3. In the **finally** part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try
    { use the string list }
  finally
    TempList.Free;{ destroy the list object }
  end;
end;
```

Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

1 In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.

2 Write an event handler for the main form's *constructor*, which executes before the form appears. It should create a string list and assign it to the field you declared in the first step.

3 Write an event handler that frees the string list for the form's *OnClose* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
{For CLX: uses SysUtils, Classes, QGraphics, QControls, QForms, Qialogs;}

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ClickList := TStringList.Create;{ construct the list }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.LOG'));{ save the list }
  ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;

end.
```

Manipulating strings in a list

Operations commonly performed on string lists include:

- Counting the strings in a list
- Accessing a particular string
- Finding the position of a string in the list
- Iterating through strings in a list
- Adding a string to a list
- Moving a string within a list
- Deleting a string from a list
- Copying a complete string list

Counting the strings in a list

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

Accessing a particular string

The *Strings* array property contains the strings in the list, referenced by a zero-based index. Because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
StringList1.Strings[0] := 'This is the first string.';
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
```

Locating items in a string list

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the *Items* of a list box:

```
if FileListBox1.Items.IndexOf('WIN.INI') > -1 ...
```

Iterating through strings in a list

To iterate through the strings in a list, use a **for** loop that runs from zero to *Count* – 1.

This example converts each string in a list box to uppercase characters.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Index: Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
end;
```

Adding a string to a list

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string “Three” the third string in a list, you would use:

```
Insert(2, 'Three');
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2);  { append the strings from StringList2 to StringList1 }
```

Moving a string within a list

To move a string in a string list, call the *Move* method, passing two parameters: the current index of the string and the index you want assigned to it. For example, to move the third string in a list to the fifth position, you would use:

```
Move(2, 4)
```

Deleting a string from a list

To delete a string from a string list, call the list’s *Delete* method, passing the index of the string you want to delete. If you don’t know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

This example uses *IndexOf* and *Delete* to find and delete a string:

```
with ListBox1.Items do
begin
  BIndex := IndexOf('bureaucracy');
  if BIndex > -1 then
    Delete(BIndex);
end;
```

Copying a complete string list

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
Memo1.Lines.Assign(ComboBox1.Items);    { overwrites original strings }
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items);   { appends strings to end }
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you assign one string-list variable to another—

```
StringList1 := StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

Associating objects with a string list

In addition to the strings stored in its *Strings* property, a string list can maintain references to objects, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear*, and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

Windows registry and INI files

The Windows system registry is a hierarchical database where applications store configuration information. The VCL class *TRegistry* supplies methods that read and write to the registry.

Until Windows 95, most applications stored configuration information in initialization files, usually named with the extension .INI. The VCL provides the following classes to facilitate maintenance and migration of programs that use INI files:

- *TRegistry* to work with the registry (VCL only).
- *TIniFile* (VCL only) or *TMemIniFile* to work with INI files.
- *TRegistryIniFile* when you want to work with both the registry and INI files (VCL only). *TRegistryIniFile* has properties and methods similar to those of *TIniFile*, but it reads and writes to the system registry. By using a variable of type *TCustomIniFile* (the common ancestor of *TIniFile*, *TMemIniFile*, and *TRegistryIniFile*), you can write generic code that accesses either the registry or an INI file, depending on where it is called.

Only *TMemIniFile* can be used in cross-platform programming.

Using TIniFile (VCL only)

The INI file format is still popular, many of the Delphi configuration files (such as the DSK Desktop settings file) are in this format. Because this file format was and is prevalent, VCL provides a class to make reading and writing these files very easy. *TIniFile* is not available for cross-platform programming.

When you instantiate the *TIniFile* object, you pass as a parameter to the constructor the name of the INI file. If the file does not exist, it is automatically created. You are then free to read values using *ReadString*, *ReadInteger*, or *ReadBool*. Alternatively, if you want to read an entire section of the INI file, you can use the *ReadSection* method. Similarly, you can write values using *WriteBool*, *WriteInteger*, or *WriteString*.

Each of the Read routines takes three parameters. The first parameter identifies the section of the INI file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the INI file. Similarly, the Write routines will create the section and/or value if they do not exist. The example code creates an INI file the first time it is run that looks like this:

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

On subsequent execution of this application, the INI values are read in during creation of the form and written back out in the *OnClose* event.

Using TRegistry

Most 32-bit applications store their information in the registry instead of INI files because the registry is hierarchical, more robust, and doesn't suffer from the size limitations of INI files. The *TRegistry* object contains methods to open, close, save, move, copy, and delete keys.

TRegistry is not available for cross-platform programming.

For more information, see the *TRegistry* topic in the online help.

Using TRegIniFile

If you are accustomed to using INI files and want to move your configuration information to the registry instead, you can use the *TRegIniFile* class. *TRegIniFile* is designed to make registry entries look like INI file entries. All the methods from *TIniFile* (read and write) exist in *TRegIniFile*.

When you construct a *TRegIniFile* object, the parameter you pass (the filename for an *IniFile* object) becomes a key value under the user key in the registry, and all sections and values branch from that root. In fact, this object simplifies the registry interface considerably, so you may want to use it instead of the *TRegistry* component even if you aren't porting existing code.

TRegIniFile is not available for cross-platform programming.

For more information, see the *TRegIniFile* topic in the VCL online reference.

Creating drawing spaces

The *TCanvas* encapsulates a Windows device context in the VCL and a paint device (Qt painter) in CLX. which handles all drawing for both forms, visual containers (such as panels) and the printer object (covered in"Printing" on page 3-54 ").

Using the canvas object, you no longer have to worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you.

TCanvas includes a large number of primitive graphics routines to draw lines, shapes, polygons, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to the middle of the form and outputs some raw text onto the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pen.Color := clBlue;
  Canvas.MoveTo( 10, 10 );
  Canvas.LineTo( 100, 100 );
  Canvas.Brush.Color := clBtnFace;
  Canvas.Font.Name := 'Arial';
  Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,'This is the end of the line' );
end;
```

In Windows applications, the *TCanvas* object also protects you against common Windows graphics errors, such as restoring device contexts, pens, brushes, and so on to the value they had before the drawing operation. *TCanvas* is used everywhere in Delphi that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

See *TCanvas* in the online help reference for a complete listing of properties and methods.

Printing

The VCL *TPrinter* object encapsulates details of Windows printers. To get a list of installed and available printers, use the *Printers* property. The CLX *TPrinter* object is a paint device that paints on a printer. It generates postscript and sends that to lpr, lp, or another print command.

Both printer objects use a *TCanvas* (which is identical to the form's *TCanvas*) which means that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method.

This example uses a button and a memo on a form. When the user clicks the button, the content of the memo is printed with a 200-pixel border around the page.

To run this example successfully, add Printers to your **uses** clause.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRect;
  i: Integer;
begin
  with Printer do
    begin
      r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
      BeginDoc;
      for i := 0 to Memo1.Lines.Count do
       Canvas.TextOut(200,200 + (i *
Canvas.TextHeight(Memo1.Lines.Strings[i])),
                                   Memo1.Lines.Strings[i]);
      Canvas.Brush.Color := clBlack;
      Canvas.FrameRect(r);
      EndDoc;
    end;
end;
```

For more information on the use of the *TPrinter* object, look in the online help under *TPrinter*.

Using streams

Streams are just ways of reading and writing data. Steams provide a common interface for reading and writing to different media such as memory, strings, sockets, and blob streams.

In the following streaming example, one file is copied to another one using streams. The application includes two edit controls (From and To) and a Copy File button.

```
procedure TForm1.CopyFileClick(Sender: TObject);
var
  stream1, stream2:TStream;
begin
  stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
  try
    stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
    try
      stream2.CopyFrom(Stream1,Stream1.Size);
    finally
      stream2.Free;
  finally
    stream1.Free
end;
```

Use specialized stream objects to read or write to storage media. Each descendant of *TStream* implements methods for accessing a particular medium, such as disk files, dynamic memory, and so on. *TStream* descendants include *TFileStream*, *TStringStream*, and *TMemoryStream*. In addition to methods for reading and writing, these objects permit applications to seek to an arbitrary position in the stream. Properties of *TStream* provide information about the stream, such as size and current position.

Chapter

4

Common programming tasks

This chapter discusses how to perform some of the common programming tasks in Delphi:

- Understanding classes
- Defining classes
- Handling exceptions
- Using interfaces
- Defining custom variants
- Working with strings
- Working with files
- Converting measurements

Understanding classes

A *class* is an abstract definition of properties, methods, events, and class members (such as variables local to the class). When you create an instance of a class, this instance is called an object. The term object is often used more loosely in the Delphi documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class.

Although Delphi includes many classes in its object hierarchy, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants. A class type declaration contains three possible sections that control the accessibility of its fields and methods:

```
Type
  TClassName = Class(TObject)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

- The public section declares fields and methods with no access restrictions; class instances and descendant classes can access these fields and methods.
- The protected section includes fields and methods with some access restrictions; descendant classes can access these fields and methods.
- The private section declares fields and methods that have rigorous access restrictions; they cannot be accessed by class instances or descendant classes.

The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior.

The general syntax of a descendant class is as follows:

```
Type
  TClassName = Class (TParentClass)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

If no parent class name is specified, the class inherits directly from *TObject*. *TObject* defines only a handful of methods, including a basic constructor and destructor.

For more information about the syntax, language definitions, and rules for classes, see the *Object Pascal Language Guide* online Help on Class types.

Defining classes

Delphi allows you to declare classes that implement the programming features you need to use in your application. Some versions of Delphi include a feature called class completion that simplifies the work of defining and implementing new classes by generating skeleton code for the class members you declare.

To define a class,

1 In the IDE, start with a project open and choose File | New | Unit to create a new unit where you can define the new class.

2 Add the **uses** clause and **type** section to the **interface** section.

3 In the **type** section, write the class declaration. You need to declare all the member variables, properties, methods, and events.

```
TMyClass = class; {This implicitly descends from TObject}
public
  .
  .
  .
  .
  .
  .
private
  .
  .
  .
published {If descended from TPersistent or below}
  .
  .
  .
```

Note The object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {$M+} compiler directive, or descend from *TPersistent* or below.

If you want the class to descend from a specific class, you need to indicate that class in the definition:

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

For example:

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

If your version of Delphi includes class completion: place the cursor within a method definition in the **interface** section and press Ctrl+Shift+C (or right-click and select Complete Class at Cursor). Delphi completes any unfinished property declarations and creates the empty methods you need in the **implementation** section. (If you do not have class completion, you'll need to write the code yourself, completing property declarations and writing the methods.)

Given the example above, if you have class completion, Delphi adds **read** and **write** specifiers to your interface declaration, including any supporting fields or methods:

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

It also adds the following code to the **implementation** section of the unit.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
```

```
end;
procedure TMyButton.SetSize(const Value: Integer);
begin
   FSize := Value;
end;
```

4 Fill in the methods. For example, to make it so the button beeps when you call the DoSomething method, add the Beep between **begin** and **end**.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
  Beep;
end;

procedure TMyButton.SetSize(const Value: Integer);
begin
  if fsize < > value then
  begin
    FSize := Value;
    DoSomething;
  end;
end;
```

Note that the button also beeps when you call SetSize to change the size of the button.

For more information about the syntax, language definitions, and rules for classes and methods, see the *Object Pascal Language Guide* online Help on Class types and methods.

Handling exceptions

Delphi provides a mechanism to handle errors in a consistent manner. Exception handling allows the application to recover from errors if possible and to shut down if need be, without losing data or resources. Error conditions in Delphi are indicated by exceptions. This section describes the following tasks for using exceptions to create safe applications:

- Protecting blocks of code
- Protecting resource allocations
- Handling RTL exceptions
- Handling component exceptions
- Exception handling with external sources
- Silent exceptions
- Defining your own exceptions

Protecting blocks of code

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application will present a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

To protect blocks of code you need to understand

- Responding to exceptions
- Exceptions and the flow of control
- Nesting exception responses

Responding to exceptions

When an error condition occurs, the application raises an exception, meaning it creates an exception object. Once an exception is raised, your application can execute cleanup code, handle the exception, or both.

Executing cleanup code

The simplest way to respond to an exception is to guarantee that some cleanup code is executed. This kind of response doesn't correct the condition that caused the error but lets you ensure that your application doesn't leave its environment in an unstable state. You typically use this kind of response to ensure that the application frees allocated resources, regardless of whether errors occur.

Handling an exception

This is a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as “File not found,” are easy to correct and retry, while others, such as running out of memory, might be more difficult for the application or the user to correct.

To handle exceptions effectively, you need to understand the following:

- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

Exceptions and the flow of control

Object Pascal makes it easy to incorporate error handling into your applications because exceptions don't get in the way of the normal flow of your code. In fact, by moving error checking and error handling out of the main flow of your algorithms, exceptions can simplify the code you write.

When you declare a protected block, you define specific responses to exceptions that might occur within that block. When an exception occurs in that block, execution immediately jumps to the response you defined, then leaves the block.

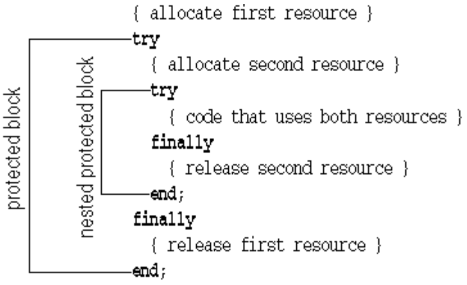
Example The following code that includes a protected block. If any exception occurs in the protected block, execution jumps to the exception-handling part, which beeps. Execution resumes outside the block.

```
try
  AssignFile(F, FileName);
  Reset(F);
  ⋮
except
  on Exception do Beep;
end;
  ⋮ { execution resumes here, outside the protected block }
```

Nesting exception responses

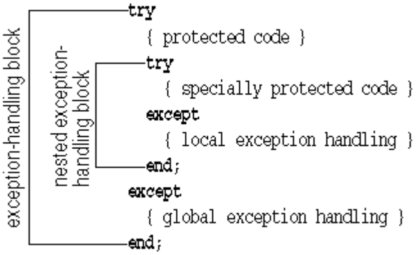
Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks of code inside other blocks, you can customize responses even within blocks that already contain customized responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:



```
{ allocate first resource }
try
  { allocate second resource }
  try
    { code that uses both resources }
  finally
    { release second resource }
  end;
finally
  { release first resource }
end;
```

You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:



```
try
  { protected code }
  try
    { specially protected code }
  except
    { local exception handling }
  end;
except
  { global exception handling }
end;
```

You can also mix different kinds of exception-response blocks, nesting resource protections within exception handling blocks and vice versa.

Protecting resource allocations

One key to having a robust application is ensuring that if it allocates resources, it also releases them, even if an exception occurs. For example, if your application allocates memory, you need to make sure it eventually releases the memory, too. If it opens a file, you need to make sure it closes the file later.

Keep in mind that exceptions don't come just from your code. A call to an RTL routine, for example, or another component in your application might raise an exception. Your code needs to ensure that if these conditions occur, you release allocated resources.

To protect resources effectively, you need to understand the following:

- What kind of resources need protection?
- Creating a resource protection block

What kind of resources need protection?

Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are:

- Files
- Memory
- Windows resources (VCL only)
- Objects

Example The following event handler allocates memory, then generates an error, so it never executes the code to free the memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an error }
  FreeMem(APointer, 1024);{ it never gets here }
end;
```

Although most errors are not that obvious, the example illustrates an important point: When the division-by-zero error occurs, execution jumps out of the block, so the *FreeMem* statement never gets to free the memory.

To guarantee that the *FreeMem* gets to free the memory allocated by *GetMem*, you need to put the code in a resource-protection block.

Creating a resource protection block

To ensure that you free allocated resources, even in case of an exception, you embed the resource-using code in a protected block, with the resource-freeing code in a special part of the block. Here's an outline of a typical protected resource allocation:

```
{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;
```

The key to the **try..finally** block is that the application always executes any statements in the **finally** part of the block, even if an exception occurs in the protected block. When any code in the **try** part of the block (or any routine called by code in the **try** part) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the **finally** part, which is called the cleanup code. After the finally part is executed, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the **try** part.

Example The following code illustrates an event handler that allocates memory and generates an error, but still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;

begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an error }
    finally
      FreeMem(APointer, 1024);{ execution resumes here, despite the error }
    end;
  end;
```

The statements in the **finally** block do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the **finally** block.

Handling RTL exceptions

When you write code that calls routines in the runtime library (RTL), such as mathematical functions or file-handling procedures, the RTL reports errors back to your application in the form of exceptions. By default, RTL exceptions generate a message that the application displays to the user. You can define your own exception handlers to handle RTL exceptions in other ways.

There are also silent exceptions that do not, by default, display a message.

RTL exceptions are handled like any other exceptions. To handle RTL exceptions effectively, you need to understand the following:

- What are RTL exceptions?
- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

What are RTL exceptions?

The runtime library's exceptions are defined in the *SysUtils* unit, and they all descend from a generic exception-object type called Exception. Exception provides the string for the message that RTL exceptions display by default.

Several kinds of exceptions can be raised by the RTL, as described in the following table.

Table 4.1 RTL exceptions

Error type	Cause	Meaning
Input/output	Error accessing a file or I/O device	Most I/O exceptions are related to error codes returned when accessing a file.
Heap	Error using dynamic memory	Heap errors can occur when there is insufficient memory available, or when an application disposes of a pointer that points outside the heap.
Integer math	Illegal operation on integer-type expressions	Errors include division by zero, numbers or expressions out of range, and overflows.

Table 4.1 RTL exceptions (continued)

Error type	Cause	Meaning
Floating-point math	Illegal operation on real-type expressions	Floating-point errors can come from either a hardware coprocessor or the software emulator. Errors include invalid instructions, division by zero, and overflow or underflow.
Typecast	Invalid typecasting with the **as** operator	Objects can only be typecast to compatible types.
Conversion	Invalid type conversion	Type-conversion functions such as IntToStr, StrToInt, and StrToFloat raise conversion exceptions when the parameter cannot be converted to the desired type.
Hardware	System condition	Hardware exceptions indicate that either the processor or the user generated some kind of error condition or interruption, such as an access violation, stack overflow, or keyboard interrupt.
Variant	Illegal type coercion	Errors can occur when referring to variants in expressions where the variant cannot be coerced into a compatible type.

For a list of the RTL exception types, see the code in the *SysUtils* unit.

Creating an exception handler

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. In cross-platform programming, it is very rare that you will need to write an exception handler. Most exceptions can be handled using **try..finally** blocks as described in "Protecting blocks of code" on page 4-4 and "Protecting resource allocations" on page 4-7.

To define an exception handler, embed the code you want to protect in an exception-handling block and specify the exception handling statements in the **except** part of the block. Here is an outline of a typical exception-handling block:

```
try
  { statements you want to protect }
except
  { exception-handling statements }
end;
```

The application executes the statements in the **except** part only if an exception occurs during execution of the statements in the **try** part. Execution of the **try** part statements includes routines called by code in the **try** part. That is, if code in the **try** part calls a routine that doesn't define its own exception handler, execution returns to the exception-handling block, which handles the exception.

When a statement in the **try** part raises an exception, execution immediately jumps to the **except** part, where it steps through the specified exception-handling statements, or exception handlers, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

Exception handling statements

Each **on** statement in the **except** part of a **try..except** block defines code for handling a particular kind of exception. The form of these exception-handling statements is as follows:

```
on <type of exception> do <statement>;
```

Example You can define an exception handler for division by zero to provide a default result:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems;{ handle the normal case }
  except
    on EDivByZero do Result := 0;{ handle the exception only if needed }
  end;
end;
```

Note that this is clearer than having to test for zero every time you call the function. Here's an equivalent function that doesn't take advantage of exceptions:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  if NumberOfItems <> 0 then{ always test }
    Result := Sum div NumberOfItems{ use normal calculation }
  else Result := 0;{ handle exceptional case }
end;
```

The difference between these two functions really defines the difference between programming with exceptions and programming without them. This example is quite simple, but you can imagine a more complex calculation involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid.

By using exceptions, you can spell out the “normal” expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every single time to make sure you're allowed to proceed with each step in the calculation.

Using the exception instance

Most of the time, an exception handler doesn't need any information about an exception other than its type, so the statements following on..do are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of on..do that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance.

Example If you create a new project that contains a single form, you can add a scroll bar and a command button to the form. Double-click the button and add the following line to its click-event handler:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

That line raises an exception because the maximum value of a scroll bar must always exceed the minimum value. The default exception handler for the application opens a dialog box containing the message in the exception object. You can override the exception handling in this handler and create your own message box containing the exception's message string:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

The temporary variable (E in this example) is of the type specified after the colon (*EInvalidOperation* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

Note Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

Scope of exception handlers

You do not need to provide handlers for every kind of exception in every block. In fact, you only need handlers for exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains the block (or returns to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

Providing default exception handlers

You can provide a single default exception handler to handle any exceptions you haven't provided specific handlers for. To do that, you add an else part to the **except** part of the exception-handling block:

```
try
  { statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from the containing block.

Caution It is not advisable to use this all-encompassing default exception handler. The **else** clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so use an enclosing **try..finally** block:

```
try
  try
  { statements }
  except
    on ESomething do { specific exception-handling code };
  end;
finally
  {cleanup code };
end;
```

For another approach to augmenting exception handling, see Reraising the exception.

Handling classes of exceptions

Because exception objects are part of a hierarchy, you can specify handlers for entire parts of the hierarchy by providing a handler for the exception class from which that part of the hierarchy descends.

Example The following block outlines an example that handles all integer math exceptions specially:

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

You can still specify specific handlers for more specific exceptions, but you need to place those handlers above the generic handler, because the application searches the handlers in the order they appear in, and executes the first applicable handler it finds. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError*.

Reraising the exception

Sometimes when you handle an exception locally, you actually want to augment the handling in the enclosing block, rather than replacing it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond.

Example When an exception occurs, you might want to display a message to the user or record the error in a log file, then proceed with the standard handling. To do that, you declare a local exception handler that displays the message then calls the reserved word raise. This is called reraising the exception, as shown in this example:

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
    begin
      { handling for only the special statements }
      raise;{ reraise the exception }
    end;
  end;
except
  on ESomething do ...;{ handling you want in all cases }
end;
```

If code in the { statements } part raises an *ESomething* exception, only the handler in the outer **except** part executes. However, if code in the { special statements } part raises *ESomething*, the handling in the inner **except** part is executed, followed by the more general handling in the outer **except** part.

By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

Handling component exceptions

Delphi's components raise exceptions to indicate error conditions. Most component exceptions indicate programming errors that would otherwise generate a runtime error. The mechanics of handling component exceptions are no different than handling RTL exceptions.

Example A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a “List index out of bounds” exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption := ListBox1.Items[3];{ set form caption to fourth string in list box }
  except
    on EStringListError do
      MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
  end;
end;
```

If you click the button once, the list box has only three strings, so accessing the fourth string (Items[3]) raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

Exception handling with external sources

HandleException provides default handling of exceptions for the application. Normally when developing cross-platform applications, you do not need to call *TApplication.HandleException*. However, you may need it when writing shared object files or callback functions. You can use *TApplication.HandleException* to block an exception from escaping from your code particularly when the code is being called from an external source that does not support exceptions.

For example, if an exception passes through all the **try** blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating that an error has occurred. You can use *HandleException* in this fashion:

```
try
  { statements }
except
  Application.HandleException(Self);
end;
```

For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. Therefore, if you want to both handle the exception, and provide this default behavior as the built-in components do, you can add a call to *HandleException* to your code:

```
try
  { special statements }
except
  on ESomething do
  begin
    { handling for only the special statements }
    Application.HandleException(Self);{ call HandleException }
  end;
end;
```

Note Do not call *HandleException* from within a thread's exception handling code.

For more information, search for exception handling routines in the Help index.

Silent exceptions

Delphi applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the

Break or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for Delphi VCL and CLX applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

Note For console applications, an error-message dialog is displayed on any unhandled *EAbort* exceptions.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which will break out of the current operation without displaying an error message.

Example The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 10 do{ loop ten times }
  begin
    ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
    if I = 7 then Abort;{ abort after the seventh one }
  end;
end;
```

Defining your own exceptions

In addition to protecting your code from exceptions generated by the runtime library and various components, you can use the same mechanism to manage exception conditions in your own code.

To use exceptions in your code, you need to complete these steps:

- Declaring an exception object type
- Raising an exception

Declaring an exception object type

Because exceptions are objects, defining a new kind of exception is as simple as declaring a new object type. Although you can raise any object instance as an exception, the standard exception handlers handle only exceptions descended from Exception.

As a convention, new exception types should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by a specific exception handler for that exception, one of the standard handlers will handle it instead.

Example For example, consider the following declaration:

```
type
  EMyException = class(Exception);
```

If you raise *EMyException* but don't provide a specific handler for it, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

Raising an exception

To indicate a disruptive error condition in an application, you can raise an exception that involves constructing an instance of that type and calling the reserved word **raise**.

To raise an exception, call the reserved word raise, followed by an instance of an exception object. This allows you to establish an exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

Raising an exception address set the *ErrorAddr* variable in the *System* unit to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user where the error occurred. You can also specify a value in the raise clause which will appear in *ErrorAddr* when an exception occurs.

Warning Do not assign a value to *ErrorAddr* yourself. It is intended as read-only.

To specify an error address for an exception, add the reserved word **at** after the exception instance, followed by an address expression such as an identifier.

For example, given the following declaration,

```
type
  EPasswordInvalid = class(Exception);
```

you can raise a “password invalid” exception at any time by calling raise with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered');
```

Using interfaces

Delphi’s **interface** keyword allows you to create and use interfaces in your application. Interfaces are a way extending the single-inheritance model of Object Pascal by allowing a single class to implement more than one interface, and by allowing several classes descended from different bases to share the same interface. Interfaces are useful when the same sets of operations, such as streaming, are used across a broad range of objects. Interfaces are also a fundamental aspect of the COM (the Component Object Model) and CORBA (Common Object Request Broker Architecture) distributed object models.

Interfaces as a language feature

An interface is like a class that contains only abstract methods and a clear definition of their functionality. Strictly speaking, interface method definitions include the number and types of their parameters, their return type, and their expected behavior. Interface methods are usually named to indicate the purpose of the interface. It is the convention to name interfaces according to their behavior and to preface them with a capital *I*. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file.

An interface has the following syntax:

```
IMyObject = interface
  procedure MyProcedure;
end;
```

A simple example of declaring an interface is:

```
type
IEdit = interface
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

Like abstract classes, interfaces themselves can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, you must define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

Implementing interfaces across the hierarchy

Using interfaces offers a design approach to separating the way a class is used from the way it is implemented. Two classes can implement the same interface without requiring that they descend from the same base class. This polymorphic invocation of the same methods on unrelated objects is possible as long as the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from say, *TFigure* which implemented a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. The above *IPaint* would be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

which makes sense for the rectangle to support but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern used to fill the circle without having to add rotation to the simple circle.

Note For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IInterface* that manage reference counting. For more information, see “Implementing IInterface” on page 4-20 and “Memory management of interface objects” on page 4-24.

Using interfaces with procedures

Interfaces also allow you to write generic procedures that can handle objects without requiring the objects to descend from a particular base class. Using the above *IPaint* and *IRotate* interfaces you can write the following procedures,

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate. This allows the above generic procedures to be used more often than if they were written to only work against a *TFigure* class.

For details about the syntax, language definitions and rules for interfaces, see the *Object Pascal Language Guide* online Help section on Object interfaces.

Implementing IInterface

All interfaces derive either directly or indirectly from the *IInterface* interface. This interface provides the essential functionality of an interface, that is, dynamic querying and lifetime management. This functionality is established in the three *IInterface* methods:

- *QueryInterface* provides a method for dynamically querying a given object and obtaining interface references for the interfaces the object supports.
- *_AddRef* is a reference counting method that increments the count each time the call to *QueryInterface* succeeds. While the reference count is nonzero the object must remain in memory.
- *_Release* is used with *_AddRef* to enable an object to track its own lifetime and to determine when it is safe to delete itself. Once the reference count reaches zero, the object is freed from memory.

Every class that implements interfaces must implement the three *IInterface* methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of life-time management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

TInterfacedObject

Delphi defines a simple class, *TInterfacedObject*, that serves as a convenient base because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;
```

Because it implements the methods of *IInterface*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see “Memory management of interface objects” on page 4-24, which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

Using the as operator

Classes that implement interfaces can use the **as** operator for dynamic binding on the interface. In the following example:

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;

begin
  X := P as IPaint;
{ statements }
end;
```

the variable *P* of type *TInterfacedObject*, can be assigned to the variable *X*, which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of *P*'s *IInterface* interface. This is because the compiler cannot tell from *P*'s declared type whether *P*'s instance actually supports *IPaint*. At runtime, *P* either resolves to an *IPaint* reference or an exception is raised. In either case, assigning *P* to *X* will not generate a compile-time error as it would if *P* was of a class type that did not implement *IInterface*.

When you use the **as** operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IInterface*: Although all interfaces derive from *IInterface*, it is not sufficient, if you want to use the **as** operator, for a class to simply implement the methods of *IInterface*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IInterface* in its interface list.
- Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the **as** operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the *Ctrl+Shift+G* editor shortcut key.

Reusing code and delegation

One approach to reusing code with interfaces is to have an object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, Object Pascal has a keyword **implements**, that makes if easy to write code to delegate all or part of the implementation of an interface to a subobject. Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object contains an inner object that implements interfaces which are exposed only by the outer object. The VCL and CLX have classes that support aggregation.

Using implements for delegation

Many classes have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword **implements** to specify that the methods of that interface are delegated to the object or interface reference which is the property instance. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list.

By default using the keyword **implements** delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods as a way of overriding this default behavior.

The following example uses the implements keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```
unit cadapt;

type
IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;

IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;

{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
  TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

For more information about the syntax, implementation details, and language rules of the **implements** keyword, see the *Object Pascal Language Guide* online Help section on Object interfaces.

Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that define the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. The only requirement is that it implement *IInterface*. The inner object, or objects, can implement one or more interfaces, however only the outer object exposes the interfaces. These include both the interfaces it implements and the ones implemented by its contained objects. Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The implementation model for aggregation defines explicit rules for implementing *IInterface* using delegation. The inner object must implement an *IInterface* on itself, that controls the inner object's reference count. This implementation of *IInterface* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *IInterface*. The inner object also implements a second *IInterface* for all the interfaces it implements. These are the interfaces exposed by the outer object. This second *IInterface* delegates calls to *QueryInterface*, *AddRef*, and *Release* to the outer object. The outer *IInterface* is referred to as the "controlling Unknown."

Refer to the MS online help for the rules about creating an aggregation. When writing your own aggregation classes, you can also refer to the implementation details of *IInterface* in *TComObject*. *TComObject* is a COM class that supports aggregation. If you are writing COM applications, you can also use *TComObject* directly as a base class.

Memory management of interface objects

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *_AddRef* and *_Release* methods of *IInterface* provide a way to implement this lifetime management. *_AddRef* and *_Release* track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero.

If you are creating COM objects for distributed applications (in the Windows environment only), then you should strictly adhere to the reference counting rules. However, if you are using interfaces only internally in your application, then you have a choice that depends upon the nature of your object and how you decide to use it.

Using reference counting

Delphi provides most of the *IInterface* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from these classes. *TInterfacedObject* is the non-CoClass that provides this behavior. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);

function test_func()
var
```

```
  y: ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
  beep(y); // the act of calling the beep function increments the reference count
  // and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
  // and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

Note In the examples above, the *beep* procedure, as it is declared, increments the reference count (call *_AddRef*) on the parameter, whereas either of the following declarations do not:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting because the lifetime of the object is not dictated by its interfaces.

Not using reference counting

If your object is a component or a control that is owned by another component, then your object is part of a different memory management system that is based in *TComponent*. You should not mix the object lifetime management approaches of VCL or CLX components and interface reference counting. If you want to create a component that supports interfaces, you can implement the *IInterface _AddRef* and *_Release* methods as empty functions to bypass the interface reference counting mechanism:

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;

function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object.

Note that, because you do implement *QueryInterface*, you can still use the **as** operator for interfaces on components, as long as you create an interface identifier (IID). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the “controlling Unknown.” It is at the level of the outer, component object that the decision is made to circumvent the *_AddRef* and *_Release* methods, and to handle memory management via the component-based approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a component as its containing outer object.

Note The “controlling Unknown” is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. *IUnknown* is the same as *IInterface*, but is used instead in COM-based applications (Windows only). For more information distinguishing the various implementations of the *IUnknown* or *IInterface* interface by the inner and outer objects, see “Aggregation” on page 4-23 and the Microsoft online Help topics on the “controlling Unknown.”

Using interfaces in distributed applications (VCL only)

Interfaces are a fundamental element in the COM, SOAP, and CORBA distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IInterface* interface methods.

When using COM, classes and interfaces are defined in terms of *IUnknown* rather than *IInterface*. There is no semantic difference between *IUnknown* and *IInterface*, the use of *IUnknown* is simply a way to adapt Delphi interfaces to the COM definition. COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class instances via CLSIDs. The CLSIDs are used to register and manipulate COM classes. COM classes that have class factories and class identifiers are called CoClasses. CoClasses take advantage of the versioning capabilities of *QueryInterface*, so that when a software module is updated *QueryInterface* can be invoked at runtime to query the current capabilities of an object.

New versions of old interfaces, as well as any new interfaces or features of an object, can become immediately available to new clients. At the same time, objects retain complete compatibility with existing client code; no recompilation is necessary because interface implementations are hidden (while the methods and parameters remain constant). In COM applications, developers can change the implementation to improve performance, or for any internal reason, without breaking any client code that relies on that interface. For more information about COM interfaces, see Chapter 33, “Overview of COM technologies.”

When distributing an application using SOAP, interfaces are required to carry their own runtime type information (RTTI). The compiler only adds RTTI to an interface when it is compiled using the {$M+} switch. Such interfaces are called *invokable interfaces*. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, client applications can only call the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can’t be called by clients.

The easiest way to define invokable interfaces is to define your interface so that it descends from *IInvokable*. *IInvokable* is the same as *IInterface*, except that it is compiled using the {$M+} switch. For more information about Web Service applications that are distributed using SOAP, and about invokable interfaces, see Chapter 31, “Using Web Services.”

Another distributed application technology is CORBA. The use of interfaces in CORBA applications is mediated by stub classes on the client and skeleton classes on the server. These stub and skeleton classes handle the details of marshaling interface calls so that parameter values and return values can be transmitted correctly. Applications must use either a stub or skeleton class, or employ the Dynamic Invocation Interface (DII) which converts all parameters to special variants (so that they carry their own type information).

Defining custom variants

One powerful built-in type of the Object Pascal language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions.

By default, variants can’t hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create a descendant of the *TCustomVariantType* class that indicates how the Variant type performs standard operations.

To create a Variant type,

1. Map the storage of the variant’s data on to the *TVarData* record.
2. Declare a class that descends from *TCustomVariantType*. Implement all required behavior (including type conversion rules) in the new class.
3. Write utility methods for creating instances of your custom variant and recognizing its type.

The above steps extend the Variant type so that the standard operators work with your new type and the new Variant type can be cast to other data types. You can further enhance your new Variant type so that it supports properties and methods that you define. When creating a Variant type that supports properties or methods, you use *TInvokeableVariantType* or *TPublishableVariantType* as a base class rather than *TCustomVariantType*.

Storing a custom variant type's data

Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first Word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type.

For example, the VarConv unit defines a custom variant type that represents a measurement. The data for this type includes the units (*TConvType*) of measurement, as well as the value (a double). The VarConv unit defines its own type to represent such a value:

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  VValue: Double;
end;
```

This type is exactly the same size as the *TVarData* record. When working with a custom variant of the new type, the variant (or its *TVarData* record) can be cast to *TConvertVarData*, and the custom Variant type simply works with the *TVarData* record as if it were a *TConvertVarData* type.

Note When defining a record that maps onto the *TVarData* record in this way, be sure to define it as a packed record.

If your new custom Variant type needs more than 14 bytes to store its data, you can define a new record type that includes a pointer or object instance. For example, the VarCmplx unit uses an instance of the class *TComplexData* to represent the data in a complex-valued variant. It therefore defines a record type the same size as *TVarData* that includes a reference to a *TComplexData* object:

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Object references are actually pointers (two Words), so this type is the same size as the *TVarData* record. As before, a complex custom variant (or its *TVarData* record), can be cast to *TComplexVarData*, and the custom variant type works with the *TVarData* record as if it were a *TComplexVarData* type.

Creating a class to enable the custom variant type

Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of *TCustomVariantType*. This involves overriding the appropriate virtual methods of *TCustomVariantType*.

Enabling casting

One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts.

There are two methods for you to implement that enable the custom Variant type to perform typecasts: *Cast*, which converts another variant type to your custom variant, and *CastTo*, which converts your custom Variant type to another type of Variant.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

For example, the following *Cast* method, from the *TComplexVariantType* class uses the type Double as an intermediate type:

```
procedure TComplexVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
  LSource, LTemp: TVarData;
begin
  VarDataInit(LSource);
  try
    VarDataCopyNoInd(LSource, Source);
    if VarDataIsStr(LSource) then
      TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
    else
    begin
      VarDataInit(LTemp);
      try
        VarDataCastTo(LTemp, LSource, varDouble);
        TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
      finally
        VarDataClear(LTemp);
      end;
    end;
    Dest.VType := VarType;
  finally
    VarDataClear(LSource);
  end;
end;
```

In addition to the use of Double as an intermediate Variant type, there are a few things to note in this implementation:

- The last step of this method sets the *VType* member of the returned *TVarData* record. This member gives the Variant type code. It is set to the *VarType* property of *TComplexVariantType*, which is the Variant type code assigned to the custom variant.
- The custom variant's data (*Dest*) is typecast from *TVarData* to the record type that is actually used to store its data (*TComplexVarData*). This makes the data easier to work with.
- The method makes a local copy of the source variant rather than working directly with its data. This prevents side effects that may affect the source data.

When casting from a complex variant to another type, the *CastTo* method also uses an intermediate type of Double (for any destination type other than a string):

```
procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
  const AVarType: TVarType);
var
  LTemp: TVarData;
begin
  if Source.VType = VarType then
    case AVarType of
      varOleStr:
        VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
      varString:
        VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
    else
      VarDataInit(LTemp);
      try
        LTemp.VType := varDouble;
        LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
        VarDataCastTo(Dest, LTemp, AVarType);
      finally
        VarDataClear(LTemp);
      end;
    end
  else
    RaiseCastError;
end;
```

Note that the *CastTo* method includes a case where the source variant data does not have a type code that matches the *VarType* property. This case only occurs for empty (unassigned) source variants.

Implementing binary operations

To allow the custom variant type to work with standard binary operators (+, -, *, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the *BinaryOp* method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand.

For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
  const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
          opAdd: Variant(Left) := Variant(Left) + TComplexVarData(Right).VComplex.AsString;
        else
          RaiseInvalidOp;
        end;
```

```
          else
            if Left.VType = VarType then
              case Operator of
                opAdd:
                  TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
                opSubtract:
                  TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
                opMultiply:
                  TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
                opDivide:
                  TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
                else
                  RaiseInvalidOp;
              end
            else
              RaiseInvalidOp;
        end
      else
        RaiseInvalidOp;
    end;
```

There are several things to note in this implementation:

This method only handles the case where the variant on the right side of the operator is a custom variant that represents a complex number. If the left-hand operand is a complex variant and the right-hand operand is not, the complex variant forces the right-hand operand first to be cast to a complex variant. It does this by overriding the *RightPromotion* method so that it always requires the type in the *VarType* property:

```
function TComplexVariantType.RightPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { Complex Op TypeX }
  RequiredVarType := VarType;
  Result := True;
end;
```

The addition operator is implemented for a string and a complex number (by casting the complex value to a string and concatenating), and the addition, subtraction, multiplication, and division operators are implemented for two complex numbers using the methods of the *TComplexData* object that is stored in the complex variant's data. This is accessed by casting the *TVarData* record to a *TComplexVarData* record and using its *VComplex* member.

Attempting any other operator or combination of types causes the method to call the *RaiseInvalidOp* method, which causes a runtime error. The *TCustomVariantType* class includes a number of utility methods such as *RaiseInvalidOp* that can be used in the implementation of custom variant types.

BinaryOp only deals with a limited number of types: strings and other complex variants. It is possible, however, to perform operations between complex numbers and other numeric types. For the *BinaryOp* method to work, the operands must be cast to complex variants before the values are passed to this method. We have already seen (above) how to use the *RightPromotion* method to force the right-hand operand to be a complex variant if the left-hand operand is complex. A similar method, *LeftPromotion*, forces a cast of the left-hand operand when the right-hand operand is complex:

```
function TComplexVariantType.LeftPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { TypeX Op Complex }
  if (Operator = opAdd) and VarDataIsStr(V) then
    RequiredVarType := varString
  else
    RequiredVarType := VarType;
  Result := True;
end;
```

This *LeftPromotion* method forces the left-hand operand to be cast to another complex variant, unless it is a string and the operation is addition, in which case *LeftPromotion* allows the operand to remain a string.

Implementing comparison operations

There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method.

The *Compare* method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the VarConv unit implements the following *Compare* method:

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
  var Relationship: TVarCompareResult);
const
  CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
    (crLessThan, crEqual, crGreaterThan);
var
  LValue: Double;
  LType: TConvType;
  LRelationship: TValueRelationship;
begin
  // supports...
  //   convvar cmp number
  //     Compare the value of convvar and the given number
  //   convvar1 cmp convvar2
  //     Compare after converting convvar2 to convvar1's unit type
  //  The right can also be a string.  If the string has unit info then it is
  //    treated like a varConvert else it is treated as a double
  LRelationship := EqualsValue;
  case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                                      TConvertVarData(Left).VConvType, LValue, LType)
        else
          RaiseCastError;
      varDouble:
        LRelationship := CompareValue(TConvertVarData(Left).VValue, TVarData(Right).VDouble);
      else
        if Left.VType = VarType then
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                          TConvertVarData(Left).VConvType, TConvertVarData(Right).VValue,
                          TConvertVarData(Right).VConvType)
        else
          RaiseInvalidOp;
    end;
    Relationship := CRelationshipToRelationship[LRelationship];
  end;
```

If the custom type does not support the concept of "greater than" or "less than," only "equal" or "not equal," however, it is difficult to implement the *Compare* method, because *Compare* must return *crLessThan*, *crEqual*, or *crGreaterThan*. When the only valid response is "not equal," it is impossible to know whether to return *crLessThan* or *crGreaterThan*. Thus, for types that do not support the concept of ordering, you can override the *CompareOp* method instead.

CompareOp has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the comparison operator. Implement this method to perform the operation and return a boolean that indicates whether the comparison is *True*. You can then call the *RaiseInvalidOp* method when the comparison makes no sense.

For example, the following *CompareOp* method comes from the *TComplexVariantType* object in the VarCmplx unit. It supports only a test of equality or inequality:

```
function TComplexVariantType.CompareOp(const Left, Right: TVarData;
  const Operator: Integer): Boolean;
begin
  Result := False;
  if (Left.VType = VarType) and (Right.VType = VarType) then
    case Operator of
      opCmpEQ:
        Result := TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
      opCmpNE:
        Result := not TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Note that the types of operands that both these implementations support are very limited. As with binary operations, you can use the *RightPromotion* and *LeftPromotion* methods to limit the cases you must consider by forcing a cast before *Compare* or *CompareOp* is called.

Implementing unary operations

To allow the custom variant type to work with standard unary operators (-, not), you must override the *UnaryOp* method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand.

For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Operator of
      opNegate:
        TComplexVarData(Right).VComplex.DoNegate;
    else
      RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Note that for the logical **not** operator, which does not make sense for complex values, this method calls *RaiseInvalidOp* to cause a runtime error.

Copying and clearing custom variants

In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type.

To indicate how to copy the variant's value, implement the *Copy* method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant's value:

```
procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
  const Indirect: Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
    with TComplexVarData(Dest) do
    begin
      VType := VarType;
      VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
    end;
end;
```

Note The *Indirect* parameter in the *Copy* method signals that the copy must take into account the case when the variant holds only an indirect reference to its data.

Tip If your custom variant type does not allocate any memory to hold its data (if the data fits entirely in the *TVarData* record), your implementation of the Copy method can simply call the *SimplisticCopy* method.

To indicate how to clear the variant's value, implement the *Clear* method. As with the Copy method, the only tricky thing about doing this is ensuring that you free any resources allocated to store the variant's data:

```
procedure TComplexVariantType.Clear(var V: TVarData);
begin
  V.VType := varEmpty;
  FreeAndNil(TComplexVarData(V).VComplex);
end;
```

You will also need to implement the *IsClear* method. This way, you can detect any invalid values or special values that represent "blank" data:

```
function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
  Result := (TComplexVarData(V).VComplex = nil) or
            TComplexVarData(V).VComplex.IsZero;
end;
```

Loading and saving custom variant values

By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the *IVarStreamable* interface from Classes.pas.

IVarStreamable defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to the stream. For example, *TComplexVariantType*, in the VarCmplx unit, implements the *IVarStreamable* methods as follows:

```
procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
  with TReader.Create(Stream, 1024) do
  try
    with TComplexVarData(Dest) do
    begin
      VComplex := TComplexData.Create;
      VComplex.Real := ReadFloat;
      VComplex.Imaginary := ReadFloat;
    end;
  finally
    Free;
  end;
end;

procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream: TStream);
begin
  with TWriter.Create(Stream, 1024) do
  try
    with TComplexVarData(Source).VComplex do
    begin
      WriteFloat(Real);
      WriteFloat(Imaginary);
      end;
    finally
      Free;
    end;
  end;
```

Note how these methods create a Reader or Writer object for the *Stream* parameter to handle the details of reading or writing values.

Using the TCustomVariantType descendant

In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the VarCmplx unit:

```
initialization
  ComplexVariantType := TComplexVariantType.Create;
```

In the finalization section of the unit that defines your *TCustomVariantType* descendant, free the instance of your class. This automatically unregisters the variant type. Here is the finalization section of the VarCmplx unit:

```
finalization
  FreeAndNil(ComplexVariantType);
```

Writing utilities to work with a custom variant type

Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be.

For example, without a utility function, the only way to create an instance of your custom variant type is to use the global *VarCast* procedure on a source variant of another type. It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out the structure you defined to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:

```
function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
  TComplexVarData(Result).VType := ComplexVariantType.VarType;
  TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

This function does not actually exist in the VarCmplx unit, but is a synthesis of methods that do exist, provided to simplify the example. Note that the returned variant is cast to the record that was defined to map onto the *TVarData* structure (*TComplexVarData*), and then filled out.

Another useful utility to create is one that returns the variant type code for your new Variant type. This type code is not a constant. It is automatically generated when you instantiate your *TCustomVariantType* descendant. It is therefore useful to provide a way to easily determine the type code for your custom variant type. The following function from the VarCmplx unit illustrates how to write one, by simply returning the *VarType* property of the *TCustomVariantType* descendant:

```
function VarComplex: TVarType;
begin
  Result := ComplexVariantType.VarType;
end;
```

Two other standard utilities provided for most custom variants check whether a given variant is of the custom type and cast an arbitrary variant to the new custom type. Here is the implementation of those utilities from the VarCmplx unit:

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
  Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;

function VarAsComplex(const AValue: Variant): Variant;
begin
  if not VarIsComplex(AValue) then
    VarCast(Result, AValue, VarComplex)
  else
    Result := AValue;
end;
```

Note that these use standard features of all variants: the *VType* member of the *TVarData* record and the *VarCast* function, which works because of the methods implemented in the *TCustomVariantType* descendant for casting data.

In addition to the standard utilities mentioned above, you can write any number of utilities specific to your new custom variant type. For example, the VarCmplx unit defines a large number of functions that implement mathematical operations on complex-valued variants.

Supporting properties and methods in custom variants

Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.

Using TInvokeableVariantType

To provide support for properties an methods, the class you create to enable the new custom variant type should descend from *TInvokeableVariantType* instead of directly from *TCustomVariantType*.

TInvokeableVariantType defines four methods:

- *DoFunction*
- *DoProcedure*
- *GetProperty*
- *SetProperty*

that you can implement to support properties and methods on your custom variant type.

For example, the VarConv unit uses *TInvokeableVariantType* as the base class for *TConvertVariantType* so that the resulting custom variants can support properties. The following example shows the property getter for these properties:

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
  const V: TVarData; const Name: String): Boolean;
var
  LType: TConvType;
begin
  // supports...
  //   'Value'
  //   'Type'
  //   'TypeName'
  //   'Family'
  //   'FamilyName'
  //   'As[Type]'
  Result := True;
  if Name = 'VALUE' then
    Variant(Dest) := TConvertVarData(V).VValue
  else if Name = 'TYPE' then
    Variant(Dest) := TConvertVarData(V).VConvType
  else if Name = 'TYPENAME' then
    Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
  else if Name = 'FAMILY' then
    Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
  else if Name = 'FAMILYNAME' then
    Variant(Dest) := ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
  else if System.Copy(Name, 1, 2) = 'AS' then
  begin
    if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
                             System.Copy(Name, 3, MaxInt), LType) then
      VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
                                   TConvertVarData(V).VConvType, LType), LType)
    else
      Result := False;
  end
  else
    Result := False;
end;
```

The *GetProperty* method checks the *Name* parameter to determine what property is wanted. It then retrieves the information from the *TVarData* record of the Variant (*V*), and returns it as a Variant (*Dest*). Note that this method supports properties whose names are dynamically generated at runtime (As[Type]), based on the current value of the custom variant.

Similarly, the *SetProperty*, *DoFunction*, and *DoProcedure* methods are sufficiently generic that you can dynamically generate method names, or respond to variable numbers and types of parameters.

Using TPublishableVariantType

If the custom variant type stores its data using an object instance, then there is an easier way to implement properties, as long as they are also properties of the object that represents the variant's data. If you use *TPublishableVariantType* as the base class for your custom variant type, then you need only implement the *GetInstance* method, and all the published properties of the object that represents the variant's data are automatically implemented for the custom variants.

For example, as was seen in "Storing a custom variant type's data" on page 4-28, *TComplexVariantType* stores the data of a complex-valued variant using an instance of *TComplexData*. *TComplexData* has a number of published properties (*Real*, *Imaginary*, *Radius*, *Theta*, and *FixedTheta*), that provide information about the complex value. *TComplexVariantType* descends from *TPublishableVariantType*, and implements the *GetInstance* method to return the *TComplexData* object (in TypInfo.pas) that is stored in a complex-valued variant's *TVarData* record:

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
  Result := TComplexVarData(V).VComplex;
end;
```

TPublishableVariantType does the rest. It overrides the *GetProperty* and *SetProperty* methods to use the runtime type information (RTTI) of the *TComplexData* object for getting and setting property values.

Note For *TPublishableVariantType* to work, the object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {$M+} compiler directive, or descend from *TPersistent*.

Working with strings

Delphi has a number of different character and string types that have been introduced throughout the development of the Object Pascal language. This section is an overview of these types, their purpose, and usage. For language details, see the Object Pascal Language online Help on String types.

Character types

Delphi has three character types: *Char*, *AnsiChar*, and *WideChar*.

The *Char* character type came from standard Pascal, and was used in Turbo Pascal and then in Object Pascal. Later Object Pascal added *AnsiChar* and *WideChar* as specific character types that were used to support standards for character representation on the Windows operating system. *AnsiChar* was introduced to support an 8-bit character ANSI standard, and *WideChar* was introduced to support a 16-bit Unicode standard. The name *WideChar* is used because Unicode characters are also known as wide characters. Wide characters are two bytes instead of one, so that the character set can represent many more different characters. When *AnsiChar* and *WideChar* were implemented, *Char* became the default character type representing the currently recommended implementation. If you use *Char* in your application, remember that its implementation is subject to change in future versions of Delphi.

Note For cross-platform programming: The Linux wchar_t widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal widechars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal widechar data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

The following table summarizes these character types:

Table 4.2 Object Pascal character types

Type	Bytes	Contents	Purpose
Char	1	A single character	Default character type
AnsiChar	1	A single character	8-bit characters
WideChar	2	A single Unicode character	16-bit Unicode standard.

For more information about using these character types, see the *Object Pascal Language Guide* online Help on Character types For more information about Unicode characters, see the *Object Pascal Language Guide* online Help on About extended character sets.

String types

Delphi has three categories of types that you can use when working with strings:

- Character pointers
- String types
- String classes

This section summarizes string types, and discusses using them with character pointers. For information about using string classes, see the online Help on TStrings.

Delphi has three string implementations: short strings, long strings, and wide strings. Several different string types represent these implementations. In addition, there is a reserved word **string** that defaults to the currently recommended string implementation.

Short strings

String was the first string type used in Turbo Pascal. **String** was originally implemented as a short string. Short strings are an allocation of between 1 and 256 bytes, of which the first byte contains the length of the string and the remaining bytes contain the characters in the string:

```
S: string[0..n]// the original string type
```

When long strings were implemented, **string** was changed to a long string implementation by default and *ShortString* was introduced as a backward compatibility type. *ShortString* is a predefined type for a maximum length string:

```
S: string[255]// the ShortString type
```

The size of the memory allocated for a *ShortString* is static, meaning that it is determined at compile time. However, the location of the memory for the *ShortString* can be dynamically allocated, for example if you use a *PShortString*, which is a pointer to a *ShortString*. The number of bytes of storage occupied by a short string type variable is the maximum length of the short string type plus one. For the *ShortString* predefined type the size is 256 bytes.

Both short strings, declared using the syntax **string**[0..n], and the *ShortString* predefined type exist primarily for backward compatibility with earlier versions of Delphi and Borland Pascal.

A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive. The {$H-} state is mostly useful for using code from versions of Object Pascal that used short strings by default. However, short strings can be useful in data structures where you need a fixed-size component or in DLLs when you don't want to use the *ShareMem* unit (see also the online Help on Memory Management). You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to string[255] or *ShortString*, which are unambiguous and independent of the $H setting.

For details about short strings and the *ShortString* type, see the *Object Pascal Language Guide* online Help on Short strings.

Long strings

Long strings are dynamically allocated strings with a maximum length of 2 Gigabytes, but the practical limit is usually dependent on the amount of available memory. Like short strings, long strings use 8-bit Ansi characters and have a length indicator. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the *Length* standard function, and to set the length of a long string you must use the *SetLength* standard procedure. Long strings are also reference-counted and, like *PChars*, long strings are null-terminated. For details about the implementation of longs strings, see the *Object Pascal Language Guide* online Help on Long strings.

Long strings are denoted by the reserved word **string** and by the predefined identifier *AnsiString*. For new applications, it is recommended that you use the long string type. All components in the VCL are compiled in this state, typically using **string**. If you write components, they should also use long strings, as should any code that receives data from string-type properties. If you want to write specific code that always uses a long string, then you should use *AnsiString*. If you want to write flexible code that allows you to easily change the type as new string implementations become standard, then you should use **string**.

WideString

The *WideChar* type allows wide character strings to be represented as arrays of *WideChars*. Wide strings are strings composed of 16-bit Unicode characters. As with long strings, wide strings are dynamically allocated with a maximum length of two Gigabytes, but the practical limit is usually dependent on the amount of available memory. In Delphi, wide strings are not reference-counted. Every assignment of a wide string to a wide string var creates a copy of the string data. In Kylix, WideStrings are reference counted.

The dynamically allocated memory that contains the string is deallocated when the wide string goes out of scope. In all other respects wide strings possess the same attributes as long strings. The *WideString* type is denoted by the predefined identifier *WideString*.

Since the 32-bit version of OLE (Windows only) uses Unicode for all strings, strings must be of wide string type in any OLE automated properties and method parameters. Also, most OLE API functions use null-terminated wide strings.

For more information, see the *Object Pascal Language Guide* topic on WideString.

PChar types

A *PChar* is a pointer to a null-terminated string of characters of the type *Char*. Each of the three character types also has a built-in pointer type:

- A *PChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PAnsiChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PWideChar* is a pointer to a null-terminated string of 16-bit characters.

PChars are, with short strings, one of the original Object Pascal string types. They were created primarily as a C language and Windows API compatibility type.

OpenString

An *OpenString* is obsolete, but you may see it in older code. It is for 16-bit compatibility and is allowed only in parameters. *OpenString* was used, before long strings were implemented, to allow a short string of an unspecified length string to be passed as a parameter. For example, this declaration:

```
procedure a(v : openstring);
```

will allow any length string to be passed as a parameter, where normally the string length of the formal and actual parameters must match exactly. You should not have to use *OpenString* in any new applications you write.

Refer also to the {$P+/-} switch in “Compiler directives for strings” on page 4-49.

Runtime library string handling routines

The runtime library provides many specialized string handling routines specific to a string type. These are routines for wide strings, longs strings, and null-terminated strings (meaning *PChars*). Routines that deal with *PChar* types use the null-termination to determine the length of the string. For more details about null-terminated strings, see Working with null-terminated strings in the *Object Pascal Language Guide* or online Help.

The runtime library also includes a category of string formatting routines. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the *Low* and *High* standard functions.

Because wide strings and long strings are the commonly used types, the remaining sections discuss these routines.

Wide character routines

When working with strings you should make sure that the code in your application can handle the strings it will encounter in the various target locales. Sometimes you will need to use wide characters and wide strings. In fact, one approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. The runtime library includes the following wide character string functions for converting between standard single-byte character strings (or MBCS strings) and Unicode strings:

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

A disadvantage of working with wide characters is that Windows 95 does not support wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require tremendous amounts of extra code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

Note Typically, CLX components represent string values as wide strings.

Commonly used long string routines

The long string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

Where appropriate, the tables also provide columns indicating whether or not a routine satisfies the following criteria.

- Uses case sensitivity: If locale settings are used, it determines the definition of case. If the routine does not use locale settings, analyses are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper and lower case characters that is determined by a predefined pattern.

- Uses locale settings: Locale settings allow you to customize your application for specific locales, in particular, for Asian language environments. Most locale settings consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Routines that use the Windows locale are typically prefaced with Ansi (that is, Ansi*XXX*).
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented as a mix of one- and two-byte character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS are written parse one- and two-byte characters.

ByteType and *StrByteType* determine whether a particular byte is the lead byte of a two-byte character. Be careful when using multi-byte characters not to truncate a string by cutting a two-byte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a to a character or string. For more information about MBCS, see “Enabling application code” on page 12-2 of Chapter 12, “Creating international applications.”

Table 4.3 String comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiCompareStr	yes	yes	yes
AnsiCompareText	no	yes	yes
AnsiCompareFileName	no	yes	yes
CompareStr	yes	no	no
CompareText	no	no	no

Table 4.4 Case conversion routines

Routine	Uses locale settings	Supports MBCS
AnsiLowerCase	yes	yes
AnsiLowerCaseFileName	yes	yes
AnsiUpperCaseFileName	yes	yes
AnsiUpperCase	yes	yes
LowerCase	no	no
UpperCase	no	no

The routines used for string file names: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the Windows locale. You should always use file names that are portable because the locale (character set) used for file names can and might differ from the default user interface.

Table 4.5 String modification routines

Routine	Case-sensitive	Supports MBCS
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
StringReplace	optional by flag	yes
Trim	NA	yes
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

Table 4.6 Sub-string routines

Routine	Case-sensitive	Supports MBCS
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes
LastDelimiter	yes	yes
QuotedStr	no	no

Table 4.7 String handling routines

Routine	Case-sensitive	Supports MBCS
AnsiContainsText	no	yes
AnsiEndsText	no	no
AnsiIndexText	no	yes
AnsiMatchText	no	yes
AnsiResemblesText	no	no
AnsiStartsText	no	yes
IfThen	NA	yes
LeftStr	yes	no
RightStr	yes	no
SoundEx	NA	no
SoundExInt	NA	no
DecodeSoundExInt	NA	no
SoundExWord	NA	no
DecodeSoundExWord	NA	no
SoundExSimilar	NA	no
SoundExCompare	NA	no

Declaring and initializing strings

When you declare a long string:

```
S: string;
```

you do not need to initialize it. Long strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
S = EmptyStr;
```

or test against an empty string:

```
S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access **nil** and will result in an access violation:

```
var
  S: string;
begin
  S[i];    // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a **nil** pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle **nil**:

```
var
  S: string;   // empty string
begin
  proc(PChar(S));  // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
S := 'No longer nil';
proc(PChar(S));// proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
SetLength(S, 100);//sets the dynamic length of S to 100
proc(PChar(S));// proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, *S* is guaranteed to reference a unique string, that is a string with a reference count of one. To obtain the length of a string, use the *Length* function.

Remember when declaring a **string** that:

```
S: string[n];
```

implicitly declares a short string, not a long string of *n* length. To declare a long string of specifically *n* length, declare a variable of type **string** and use the *SetLength* procedure.

```
S: string;
SetLength(S, n);
```

Mixing and converting string types

Short, long, and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

Additional functions (*CopyQStringListToTstrings, Copy TStringsToQStringList, QStringListToTStringList*) are provided for converting underlying Qt string types and CLX string types. These functions are located in Qtypes.pas.

String to PChar conversions

Long string to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.
- Long strings are null-terminated and also contain the length of the string, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in this section.

String dependencies

Sometimes you will need convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. For example:

```
procedure my_func(x: string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

Returning a PChar local variable

A common error when working with *PChars* is to store in a data structure, or return as a value, a local variable. When your routine ends, the *PChar* will disappear because it is simply a pointer to memory, and is not a reference counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format(‘title - %d’, [n]);
  Result := PChar(s); // DON’T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

Passing a local variable as a PChar

Consider that you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:

```
// VCL version
// assume MAXSIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := GetModuleFilename(0, @buf, SizeOf(buf));// treats @buf as a PChar
  S := buf;
  //statements
end;
```

Or, for cross-platform programs, the code is nearly identical:

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
  . . .
end;

// assume MAX_SIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, @buf, SizeOf(buf));// treats @buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an **array of char** and a **string** is automatic. When *GetModuleFilename* (or *FillBuffer* in the cross-platform version) returns, the *Length* of the string correctly indicates the number of bytes written to *buf*.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an **array of char** to a **string**. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
  SetLength(S, MAX_SIZE;// when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // statements
end;
```

Compiler directives for strings

The following compiler directives affect character and string types.

Table 4.8 Compiler directives for strings

Directive	Description
{$H+/-}	A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive.
{$P+/-}	The $P directive is meaningful only for code compiled in the {$H-} state, and is provided for backwards compatibility. $P controls the meaning of variable parameters declared using the string keyword in the {$H-} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters.

{$V+/-}	The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types.
	In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:
	`var S: string[3];`
	`procedure Test(var T: string);`
	`begin`
	`  T := ‘1234’;`
	`end;`
	`begin`
	`  Test(S);`
	`end.`

{$X+/-}	The {$X+} compiler directive enables support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with *Write*, *Writeln*, *Val*, *Assign*, and *Rename* from the System unit.)

Strings and characters: related topics

The following *Object Pascal Language Guide* topics discuss strings and character sets. Also see Chapter 12, “Creating international applications.”

- About extended character sets (Discusses international character sets)
- Working with null-terminated strings (Contains information about character arrays)
- Character strings
- Character pointers
- String operators

Working with files

This section describes working with files and distinguishes between manipulating files on disk, and input/output operations such as reading and writing to files. The first section discusses the runtime library and Windows API routines you would use for common programming tasks that involve manipulating files on disk. The next section is an overview of file types used with file I/O. The last section focuses on the recommended approach to working with file I/O, which is to use file streams.

Although the Object Pascal language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications.

Note Previous versions of the Object Pascal language performed operations on files themselves, rather than on the filename parameters commonly used now. With these file types you had to locate a file and assign it to a file variable before you could, for example, rename the file.

Manipulating files

Several common file operations are built into Object Pascal's runtime library. The procedures and functions for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead. Object Pascal provides routines for most file manipulation. When it does not, alternative routines are discussed.

Caution Although the Object Pascal language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications.

Deleting a file

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm deletions of files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

DeleteFile returns *True* if it deleted the file and *False* if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

Finding a file

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminates a *FindFirst*/*FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns *True* if the file exists, *False* otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. The declaration for *TSearchRec* is:

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;//Time contains the time stamp of the file.
    Size: Integer;//Size contains the size of the file in bytes.
    Attr: Integer;//Attr represents the file attributes of the file.
    Name: TFileName;//Name contains the filename and extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;//FindData contains additional information such as
    //file creation time, last access time, long and short filenames.
  end;
```

If a file is found, the fields of the *TSearchRec* type parameter are modified to describe the found file. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Table 4.9 Attribute constants and values

Constant	Value	Description
faReadOnly	$00000001	Read-only files
faHidden	$00000002	Hidden files
faSysFile	$00000004	System files
faVolumeID	$00000008	Volume ID files
faDirectory	$00000010	Directory files
faArchive	$00000020	Archive files
faAnyFile	$0000003F	Any file

To test for an attribute, combine the value of the *Attr* field with the attribute constant with the **and** operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*: (*SearchRec.Attr* and *faHidden* > 0). Attributes can be combined by OR'ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass (*faReadOnly* or *faHidden*) the *Attr* parameter.

Example: This example uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption. Each time the user clicks the *Again* button, the next matching filename and size is displayed in the label:

```
var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\delphi6\bin\*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
end;

procedure TForm1.AgainClick(Sender: TObject);
begin
  if (FindNext(SearchRec) = 0)
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
  else
    FindClose(SearchRec);
end;
```

In cross-platform applications, you should replace any hardcoded pathnames such as c:\Program Files\delphi6\bin\*.* with the correct pathname for the system or use environment variables (on the Environment Variables page when you choose Tools | Environment Options) to represent them.

Renaming a file

To change a filename, simply use the *RenameFile* function:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

which changes a filename, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file, for example, if a file called *NewFileName* already exists, it returns *False*. For example:

```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
```

You cannot rename (move) a file across drives using *RenameFile*. You would need to first copy the file and then delete the old one.

Note *RenameFile* in the VCL is a wrapper around the Windows API *MoveFile* function, so *MoveFile* will not work across drives either.

File date-time routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or -1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, take a *Handle* type as a parameter. To get access to a Windows file *Handle* either

- Call the Windows API *CreateFile* function. *CreateFile* is a 32-bit only function that creates or opens a file and returns a *Handle* that can be used to access the file.
- Instantiate *TFileStream* to create or open a file. Then use the *Handle* property as you would a Windows' file *Handle*. See "Using file streams" on page 4-54 for more information.

Copying a file

The runtime library does not provide any routines for copying a file. However, if you are writing Windows-only applications, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the Delphi runtime library file routines, *CopyFile* takes a filename as a parameter, not a *Handle*. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the Delphi *RenameFile* function nor the Windows API *MoveFile* function can rename/move files across drives. For more information, see the Microsoft Windows online Help.

File types with file I/O

You can use three file types when working with file I/O: Pascal file types, file handles, and file stream objects. The following table summarizes these types.

Table 4.10 File types for file I/O

File type	Description
Pascal file types	In the System unit. These types are used with file variables, usually of the format "F: Text:" or "F: File". The files have three types: typed, text, and untyped. A number of file-handling routines, such as *AssignPrn* and *writeln,* use them. These file types are obsolete and are incompatible with Windows file handles. If you need to work with them, see the *Object Pascal Language Guide*.
File handles	In the Sysutils unit. A number of routines use a handle to identify the file. You get the handle when you open or create the file (for example, using FileOpen or FileCreate). Once you have the handle, there are routines to work with the contents of the file given its handle (write a line, read text, and so on). In Windows programming, the Object Pascal file handles are wrappers for the Windows file handle type. The runtime library file-handling routines that use Windows file Handles are typically wrappers around Windows API functions. For example, the FileRead calls the Windows ReadFile function. Because the Delphi functions use Object Pascal syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Using these routines is straightforward, and if you are familiar and comfortable with the Windows API file routines, you may want to use them when working with file I/O.
File streams	File streams are object instances of the *TFileStream* class used to access information in disk files. File streams are a portable and high-level approach to file I/O. *TFileStream* has a *Handle* property that lets you access the file handle. The next section discusses *TFileStream*.

Using file streams

TFileStream is a class that enables applications to read from and write to a file on disk. It is used for high-level object representations of file streams. *TFileStream* offers multiple functionality: persistence, interaction with other streams, and file I/O.

- *TFileStream* is a descendant of the stream classes. As such, one advantage of using file streams is that they inherit the ability to persistently store component properties. The stream classes work with the *TFiler* classes, *TReader,* and *TWriter*, to stream objects out to disk. Therefore, when you have a file stream, you can use that same code for the component streaming mechanism. For more information about using the component streaming system, see the online Help on the *TStream*, *TFiler*, *TReader*, *TWriter*, and *TComponent* classes.
- *TFileStream* can interact easily with other stream classes. For example, if you want to dump a dynamic memory block to disk, you can do so using a *TFileStream* and a *TMemoryStream*.
- *TFileStream* provides the basic methods and properties for file I/O. The remaining sections focus on this aspect of file streams

Creating and opening files

To create or open a file and get access to a handle for the file, you simply instantiate a *TFileStream*. This opens or creates a named file and provides methods to read from or write to it. If the file cannot be opened, *TFileStream* raises an exception.

```
constructor Create(const filename: string; Mode: Word);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode or'ed together. The open mode must be one of the following values:

Table 4.11 Open modes

Value	Meaning
fmCreate	TFileStream a file with the given name. If a file with the given name exists, open the file in write mode.
fmOpenRead	Open the file for reading only.
fmOpenWrite	Open the file for writing only. Writing to the file completely replaces the current contents.
fmOpenReadWrite	Open the file to modify the current contents rather than replace them.

The share mode can be one of the following values with the restrictions listed below:

Table 4.12 Shared modes

Value	Meaning
fmShareCompat	Sharing is compatible with the way FCBs are opened.
fmShareExclusive	Other applications can not open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.
fmShareDenyRead	Other applications can open the file for writing but not for reading.
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

Note that which share mode you can use depends on which open mode you used. The following table shows shared modes that are available for each open mode.

Table 4.13 Shared modes available for each open mode

Open Mode	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	Can't use	Can't use	Available	Can't use	Available
fmOpenWrite	Available	Available	Can't use	Available	Available
fmOpenReadWrite	Available	Available	Available	Available	Available

The file open and share mode constants are defined in the SysUtils unit.

Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. *Handle* is read-only and indicates the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a window's file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a window's file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

Reading and writing to files

TFileStream has several different methods for reading from and writing to files. These are distinguished by whether they perform the following:

- Return the number of bytes read or written.
- Require the number of bytes is known.
- Raise an exception on error.

Read is a function that reads up to *Count* bytes from the file associated with the file stream, starting at the current *Position*, into *Buffer*. *Read* then advances the current position in the file by the number of bytes actually transferred. The prototype for *Read* is

```
function Read(var Buffer; Count: Longint): Longint; override;
```

Read is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the end of file marker is encountered.

Write is a function that writes *Count* bytes from the *Buffer* to the file associated with the file stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint; override;
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered.

The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception on error (*EReadError* and *EWriteError*) while the *Read* and *Write* methods do not. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
```

```
procedure WriteBuffer(const Buffer; Count: Longint);
```

These methods call the *Read* and *Write* methods, to perform the actual reading and writing.

Reading and writing strings

If you are passing a string to a read or write function, you need to be aware of the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in only the pointer element. What you need to do is first cast the string to a *Pointer* or *PChar,* and then dereference it. For example:

```
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// this will give you garbage
  fs.Write(PChar(s)^, Length(s));// this is the correct way
end;
```

Seeking a file

Most typical file I/O mechanisms have a process of seeking a file in order to read from or write to a particular location within it. For this purpose, *TFileStream* has a *Seek* method. The prototype for *Seek* is:

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

Value	Meaning
soFromBeginning	Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be >= 0.
soFromCurrent	Offset is from the current position in the resource. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the resource. Offset must be <= 0 to indicate a number of bytes before the end of the file.

Seek resets the current *Position* of the stream, moving it by the indicated offset. *Seek* returns the new value of the *Position* property, the new current position in the resource.

File position and size

TFileStream has properties that hold the current position and size of the file. These are used by the *Seek*, read, and write methods.

The *Position* property of *TFileStream* is used to indicate the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Longint;
```

The *Size* property indicates the size in bytes of the stream. It is used as an end of file marker to truncate the file. The declaration for *Size* is:

```
property Size: Longint;
```

Size is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the file. If the *Size* of the file cannot be changed, an exception is raised. For example, trying to change the *Size* of a file that was opened in *fmOpenRead* mode raises an exception.

Copying

CopyFrom copies a specified number of bytes from one (file) stream to another.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Using *CopyFrom* eliminates the need to create, read into, write from, and free a buffer when copying data.

CopyFrom copies *Count* bytes from *Source* into the stream. *CopyFrom* then moves the current position by *Count* bytes, and returns the number of bytes copied. If *Count* is 0, *CopyFrom* sets *Source* position to 0 before reading and then copies the entire contents of *Source* into the stream. If *Count* is greater than or less than 0, *CopyFrom* reads from the current position in *Source*.

Converting measurements

The ConvUtils unit declares a general-purpose *Convert* function that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*. The units you're converting must be in the same conversion family. For information on doing conversions, see the next section Performing conversions and refer to *Convert* in the online Help.

The StdConvs unit defines several conversion families and measurement units within each family. In addition, you can create customized conversion families and associated units using the *RegisterConversionType* and *RegisterConversionFamily* functions. For information on extending conversion and conversion units, see the section Adding new measurement types and refer to Convert in the online Help.

Performing conversions

You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.

Performing simple conversions

You can use the *Convert* function to convert a measurement from one set of units to another. The *Convert* function converts between units that measure the same type of thing (distance, area, time, temperature, and so on).

To use *Convert*, you must specify the units from which to convert and to which to convert. You use the *TConvType* type to identify the units of measurement.

For example, this converts a temperature from degrees Fahrenheit to degrees Kelvin:

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

Performing complex conversions

You can also use the *Convert* function to perform more complex conversions between the ratio of two measurement types. Examples of when you might need to use this this are when converting miles per hour to meters per minute for calculating speed or when converting gallons per minute to liters per hour for calculating flow.

For example, the following call converts miles per gallon to kilometers per liter:

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

The units you're converting must be in the same conversion family (they must measure the same thing). If the units are not compatible, *Convert* raises an *EConversionError* exception. You can check whether two *TConvType* values are in the same conversion family by calling *CompatibleConversionTypes*.

The StdConvs unit defines several families of *TConvType* values. See Conversion family variables in the online Help for a list of the predefined families of measurement units and the measurement units in each family.

Adding new measurement types

If you want to perform conversions between measurement units not already defined in the StdConvs unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values.

You first need to obtain *TConvFamily* values by registering a conversion family using the *RegisterConversionFamily* function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the StdConvs unit), you can use the *RegisterConversionType* function to add the new units to the conversion family. The following examples show how to do this.

For more examples, refer to the source code for the standard conversions unit (stdconvs.pas). (Note that the source is not included in all versions of Delphi.)

Creating a simple conversion family and adding units

One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur.

To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not an exact conversion between days and months, days and years, and so on. Months have different lengths; years have correction factors for leap years, leap seconds, and so on.

If you are only using units of measurement greater than or equal to months, you can create a more accurate conversion family with years as its base unit. This example creates a new conversion family called *cbLongTime*.

Declare variables

First, you need to declare variables for the identifiers. The identifiers are used in the new LongTime conversion family, and the units of measurement that are its members:

```
var
  cbLongTime: TConvFamily;
  ltMonths: TConvType;
  ltYears: TConvType;
  ltDecades: TConvType;
  ltCenturies: TConvType;
  ltMillennia: TConvType;
```

Register the conversion family

Next, register the conversion family:

```
cbLongTime := RegisterConversionFamily ('Long Times');
```

Although an *UnregisterConversionFamily* procedure is provided, you don't need to unregister conversion families unless the unit that defines them is removed at runtime. They are automatically cleaned up when your application shuts down.

Register measurement units

Next, you need to register the measurement units within the conversion family that you just created. You use the *RegisterConversionType* function, which registers units of measurement within a specified family. You need to define the base unit which in the example is years, and the other units are defined using a factor that indicates their relation to the base unit. So, the factor for *ltMonths* is 1/12 because the base unit for the LongTime family is years. You also include a description of the units to which you are converting.

The code to register the measurement units is shown here:

```
ltMonths:=RegisterConversionType(cbLongTime,'Months',1/12);
ltYears:=RegisterConversionType(cbLongTime,'Years',1);
ltDecades:=RegisterConversionType(cbLongTime,'Decades',10);
ltCenturies:=RegisterConversionType(cbLongTime,'Centuries',100);
ltMillennia:=RegisterConversionType(cbLongTime,'Millennia',1000);
```

Use the new units

You can now use the newly registered units to perform conversions. The global *Convert* function can convert between any of the conversion types that you registered with the *cbLongTime* conversion family.

So instead of using the following *Convert* call,

```
Convert(StrToFloat(Edit1.Text),tuMonths,tuMillennia);
```

you can now use this one for greater accuracy:

```
Convert(StrToFloat(Edit1.Text),ltMonths,ltMillennia);
```

Using a conversion function

For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins.

This example, which comes from the StdConvs unit, shows how to register a conversion type by providing functions to convert to and from the base units.

Declare variables

First, declare variables for the identifiers. The identifiers are used in the *cbTemperature* conversion family, and the units of measurement are its members:

```
var
   cbTemperature: TConvFamily;
   tuCelsius: TConvType;
   tuKelvin: TConvType;
   tuFahrenheit: TConvType;
```

Note The units of measurement listed here are a subset of the temperature units actually registered in the *StdConvs* unit.

Register the conversion family

Next, register the conversion family:

```
cbTemperature := RegisterConversionFamily ('Temperature');
```

Register the base unit

Next, define and register the base unit of the conversion family, which in the example is degrees Celsius. Note that in the case of the base unit, we can use a simple conversion factor, because there is no actual conversion to make:

```
tuCelsius:=RegisterConversionType(cbTemperature,'Celsius',1);
```

Write methods to convert to and from the base unit

You need to write the code that performs the conversion from each temperature scale to and from degrees Celsius, because these do not rely on a simple conversion factor. These functions are taken from the StdConvs unit:

```
function FahrenheitToCelsius(const AValue: Double): Double;
begin
  Result := ((AValue - 32) * 5) / 9;
end;

function CelsiusToFahrenheit(const AValue: Double): Double;
begin
  Result := ((AValue * 9) / 5) + 32;
end;

function KelvinToCelsius(const AValue: Double): Double;
begin
  Result := AValue - 273.15;
end;

function CelsiusToKelvin(const AValue: Double): Double;
begin
  Result := AValue + 273.15;
end;
```

Register the other units

Now that you have the conversion functions, you can register the other measurement units within the conversion family. You also include a description of the units.

The code to register the other units in the family is shown here:

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
CelsiusToKelvin);
  tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit', FahrenheitToCelsius,
CelsiusToFahrenheit);
```

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the conversion types that you registered with the *cbTemperature* conversion family. For example the following code converts a value from degrees Fahrenheit to degrees Kelvin.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

Using a class to manage conversions

You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing.

If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros), you can't use a simple conversion factor approach to properly convert between European currencies for two reasons:

- The conversion must round to a currency-specific number of digits.
- The conversion factor approach uses an inverse factor to the one specified by the standard Euro conversions.

However, this can all be handled by the conversion functions such as the following:

```
function FromEuro(const AValue: Double, Factor, FRound): Double;
begin
  Result := RoundTo(AValue * Factor, FRound);
end;

function ToEuro(const AValue: Double, Factor): Double;
begin
  Result := AValue / Factor;
end;
```

The problem is, this approach requires extra parameters on the conversion function, which means you can't simply register the same function with every European currency. In order to avoid having to write two new conversion functions for every European currency, you can make use of the same two functions by making them the members of a class.

Creating the conversion class

The class must be a descendant of *TConvTypeFactor*. *TConvTypeFactor* defines two methods, *ToCommon* and *FromCommon*, for converting to and from the base units of a conversion family (in this case, to and from Euros). Just as with the functions you use directly when registering a conversion unit, these methods have no extra parameters, so you must supply the number of digits to round off and the conversion factor as private members of your conversion class. This is shown in the EuroConv example in the demos\ConvertIt directory (see euroconv.pas):

```
type
  TConvTypeEuroFactor = class(TConvTypeFactor)
  private
    FRound: TRoundToRange;
  public
    constructor Create(const AConvFamily: TConvFamily;
      const ADescription: string; const AFactor: Double;
      const ARound: TRoundToRange);
    function ToCommon(const AValue: Double): Double; override;
    function FromCommon(const AValue: Double): Double; override;
  end;
end;
```

The constructor assigns values to those private members:

```
constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
  const ADescription: string; const AFactor: Double;
  const ARound: TRoundToRange);
begin
  inherited Create(AConvFamily, ADescription, AFactor);
  FRound := ARound;
end;
```

The two conversion functions simply use these private members:

```
function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
  Result := SimpleRoundTo(AValue * Factor, FRound);
end;

function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
begin
  Result := AValue / Factor;
end;
```

Declare variables

Now that you have a conversion class, begin as with any other conversion family, by declaring identifiers:

```
var
  euEUR: TConvType; { EU euro }
  euBEF: TConvType; { Belgian francs }
  euDEM: TConvType; { German marks }
  euGRD: TConvType; { Greek drachmas }
  euESP: TConvType; { Spanish pesetas }
  euFFR: TConvType; { French francs }
  euIEP: TConvType; { Irish pounds }
  euITL: TConvType; { Italian lire }
  euLUF: TConvType; { Luxembourg francs }
  euNLG: TConvType; { Dutch guilders }
```

```
    euATS: TConvType; { Austrian schillings }
    euPTE: TConvType; { Portuguese escudos }
    euFIM: TConvType; { Finnish marks }
    euUSD: TConvType; { US dollars }
    euGBP: TConvType; { British pounds }
    euJPY: TConvType; { Japanese yen }
```

Register the conversion family and the other units

Now you are ready to register the conversion family and the European monetary units, using your new conversion class:

```
cbEuro := RegisterConversionFamily ('European currency');
...
// Euro's various conversion types
euEUR := RegisterEuroConversionType(cbEuro, SEURDescription, EURToEUR, EURSubUnit);
euBEF := RegisterEuroConversionType(cbEuro, SBEFDescription, BEFToEUR, BEFSubUnit);
euDEM := RegisterEuroConversionType(cbEuro, SDEMDescription, DEMToEUR, DEMSubUnit);
euGRD := RegisterEuroConversionType(cbEuro, SGRDDescription, GRDToEUR, GRDSubUnit);
euESP := RegisterEuroConversionType(cbEuro, SESPDescription, ESPToEUR, ESPSubUnit);
euFFR := RegisterEuroConversionType(cbEuro, SFFRDescription, FFRToEUR, FFRSubUnit);
euIEP := RegisterEuroConversionType(cbEuro, SIEPDescription, IEPToEUR, IEPSubUnit);
euITL := RegisterEuroConversionType(cbEuro, SITLDescription, ITLToEUR, ITLSubUnit);
euLUF := RegisterEuroConversionType(cbEuro, SLUFDescription, LUFToEUR, LUFSubUnit);
euNLG := RegisterEuroConversionType(cbEuro, SNLGDescription, NLGToEUR, NLGSubUnit);
euATS := RegisterEuroConversionType(cbEuro, SATSDescription, ATSToEUR, ATSSubUnit);
euPTE := RegisterEuroConversionType(cbEuro, SPTEDescription, PTEToEUR, PTESubUnit);
euFIM := RegisterEuroConversionType(cbEuro, SFIMDescription, FIMToEUR, FIMSubUnit);
euUSD := RegisterEuroConversionType(cbEuro, SUSDDescription,
                                    ConvertUSDToEUR, ConvertEURToUSD);
euGBP := RegisterEuroConversionType(cbEuro, SGBPDescription,
                                    ConvertGBPToEUR, ConvertEURToGBP);
euJPY := RegisterEuroConversionType(cbEuro, SJPYDescription,
                                    ConvertJPYToEUR, ConvertEURToJPY);
```

Note that *RegisterEuroConversionType* is a wrapper function that simplifies the registering of the monetary types. See the example code for details.

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the European currencies you have registered with the new cbEuro family. For example, the following code converts a value from Italian Lire to German Marks:

```
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
```

Defining data types

Object Pascal has many predefined data types. You can use these predefined types to create new types that meet the specific needs of your application. For an overview of types, see the *Object Pascal Language Guide*.

Chapter 5

Building applications, components, and libraries

This chapter provides an overview of how to use Delphi to create applications, libraries, and components.

Creating applications

The main use of Delphi is designing and building the following types of applications:

- GUI applications
- Console applications
- Service applications (for Windows applications only)
- Packages and DLLs

GUI applications generally have an easy-to-use interface. Console applications run from a console window. Service applications are run as Windows services. These types of applications compile as executables with start-up code.

You can create other types of projects such as packages and DLLs that result in creating packages or dynamically linkable libraries. These applications produce executable code without start-up code. Refer to "Creating packages and DLLs" on page 5-9.

GUI applications

A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can extend the application by calling DLLs, packages, and other support files from the executable.

Delphi offers two application UI models:

- Single document interface (SDI)
- Multiple document interface (MDI)

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE.

User interface models

Any form can be implemented as a multiple document interface (MDI) or single document interface (SDI) form. In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors. An SDI application, in contrast, normally contains a single document view. To make your form an SDI application, set the *FormStyle* property of your *Form* object to *fsNormal*.

For more information on developing the UI for an application, see Chapter 6, "Developing the application user interface."

SDI applications

To create a new SDI application,

1. Select File | New | Other to bring up the New Items dialog.
2. Click on the Projects page and select SDI Application.
3. Click OK.

By default, the *FormStyle* property of your *Form* object is set to *fsNormal*, so Delphi assumes that all new applications are SDI applications.

MDI applications

To create a new MDI application,

1. Select File | New | Other to bring up the New Items dialog.
2. Click on the Projects page and select MDI Application.
3. Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIForm*) or main form (*fsMDIChild*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form's properties.

Setting IDE, project, and compilation options

Choose Project | Options to specify various options for your project. For more information, see the online Help.

Setting default project options

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the Default box at the bottom right of the window. All new projects will use the current options selected by default.

Programming templates

Programming templates are commonly used "skeleton" structures that you can add to your source code and then fill in. Some standard code templates such as those for array, class, and function declarations, and many statements, are included with Delphi.

You can also write your own templates for coding structures that you often use. For example, if you want to use a **for** loop in your code, you could insert the following template:

```
for :=  to  do
begin

end;
```

To insert a code template in the Code editor, press *Ctrl-j* and select the template you want to use. You can also add your own templates to this collection. To add a template:

1 Select Tools | Editor Options.

2 Click the Code Insight tab.

3 In the Templates section, click Add.

4 Type a name for the template after Shortcut name and enter a brief description of the new template.

5 Add the template code to the Code text box.

6 Click OK.

Console applications

Console applications are 32-bit programs that run without a graphical interface, usually in a console window. These applications typically don't require much user input and perform a limited set of functions.

To create a new console application,

1 Choose File | New | Other and select Console Application from the New Items dialog box.

Delphi then creates a project file for this type of source file and displays the code editor.

Note When you create a new console application, the IDE does not create a new form. Only the code editor is displayed.

Service applications

Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A web, FTP, or e-mail server is an example of a service application.

To create an application that implements a Win32 service, Choose File|New, and select Service Application from the New Items page. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.

Once you have created a service application, you will see a window in the designer that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the Object Inspector. You can add additional services to your service application by choosing Service from the new items dialog. Do not add services to an application that is not a service application. While a *TService* object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.

Once your service application is built, you can install its services with the Service Control Manager (SCM). Other applications can then launch your services by sending requests to the SCM.

To install your application's services, run it using the /INSTALL option. The application installs its services and exits, giving a confirmation message if the services are successfully installed. You can suppress the confirmation message by running the service application using the /SILENT option.

To uninstall the services, run it from the command line using the /UNINSTALL option. (You can also use the /SILENT option to suppress the confirmation message when uninstalling).

Example This service has a *TServerSocket* whose port is set to 80. This is the default port for Web Browsers to make requests to Web Servers and for Web Servers to make responses to Web Browsers. This particular example produces a text document in the C:\Temp directory called WebLog*xxx*.log (where *xxx* is the ThreadID). There should be only one Server listening on any given port, so if you have a web server, you should make sure that it is not listening (the service is stopped).

To see the results: open up a web browser on the local machine and for the address, type 'localhost' (with no quotes). The Browser will time out eventually, but you should now have a file called weblog*xxx*.log in the C:\temp directory.

1 To create the example, choose File|New and select Service Application from the New Items dialog. You will see a window appear named Service1. From the Internet page of the Component palette, add a ServerSocket component to the service window (Service1).

2 Next, add a private data member of type *TMemoryStream* to the TService1 class. The interface section of your unit should now look like this:

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;

type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;

var
  Service1: TService1;
```

3 Next, select ServerSocket1, the component you added in step 1. In the Object Inspector, double click the *OnClientRead* event and add the following event handler:

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;

begin
  Buffer := nil;

while Socket.ReceiveLength > 0 do begin
    Buffer := AllocMem(Socket.ReceiveLength);
    try
      Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
      Stream.Write(Buffer^, StrLen(Buffer));
    finally
      FreeMem(Buffer);
    end;

  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
  end;
end;
```

4 Finally, select Service1 by clicking in the window's client area (but not on the ServiceSocket). In the Object Inspector, double click the *OnExecute* event and add the following event handler:

```
procedure TService1.Service1Execute(Sender: TService);
begin
  Stream := TMemoryStream.Create;
  try
```

```
    ServerSocket1.Port := 80; // WWW port
    ServerSocket1.Active := True;

    while not Terminated do begin
      ServiceThread.ProcessRequests(True);
    end;

    ServerSocket1.Active := False;
  finally
    Stream.Free;
  end;
end;
```

When writing your service application, you should be aware of:

- Service threads
- Service name properties
- Debugging services

Service threads

Each service has its own thread (*TServiceThread*), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TService OnExecute* event handler. The service thread has its own *Execute* method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests.

Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread (derived from *TThread*, not *TServiceThread*) for each request and move the implementation of that service to the new thread's *Execute* method. This allows the service thread's *Execute* loop to process new requests continually without having to wait for the service's *OnExecute* handler to finish. The following example demonstrates.

Example This service beeps every 500 milliseconds from within the standard thread. It handles pausing, continuing, and stopping of the thread when the service is told to pause, continue, or stop.

1 Choose File|New|Other and select Service Application from the New Items dialog. You will see a window appear named Service1.

2 In the interface section of your unit, declare a new descendant of *TThread* named TSparkyThread. This is the thread that does the work for your service. The declaration should appear as follows:

```
TSparkyThread = class(TThread)
    public
      procedure Execute; override;
  end;
```

3 Next, in the implementation section of your unit, create a global variable for a TSparkyThread instance:

```
var
  SparkyThread: TSparkyThread;
```

4 Add the following code to the implementation section for the TSparkyThread Execute method (the thread function):

```
procedure TSparkyThread.Execute;
begin
  while not Terminated do
  begin
    Beep;
    Sleep(500);
  end;
end;
```

5 Select the Service window (Service1), and double-click the OnStart event in the Object Inspector. Add the following OnStart event handler:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
```

6 Double-click the OnContinue event in the Object Inspector. Add the following OnContinue event handler:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
  SparkyThread.Resume;
  Continued := True;
end;
```

7 Double-click the OnPause event in the Object Inspector. Add the following OnPause event handler:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
```

8 Finally, double-click the OnStop event in the Object Inspector and add the following OnStop event handler:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

When developing server applications, choosing to spawn a new thread depends on the nature of the service being provided, the anticipated number of connections, and the expected number of processors on the computer running the service.

Service name properties

The VCL provides classes for creating service applications on the Windows platform (not available for cross-platform applications). These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This section describes the differences.

Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name* property. The following sections summarize the name properties:

TDependency properties

The *TDependency DisplayName* is both a display name and the actual name of the service. It is nearly always the same as the *TDependency Name* property.

TService name properties

The *TService Name* property is inherited from *TComponent*. It is the name of the component, and is also the name of the service. For dependencies that are services, this property is the same as the *TDependency Name* and *DisplayName* properties.

TService's *DisplayName* is the name displayed in the Service Manager window. This often differs from the actual service name (*TService.Name*, *TDependency.DisplayName*, *TDependency.Name*). Note that the *DisplayName* for the Dependency and the *DisplayName* for the Service usually differ.

Service start names are distinct from both the service display names and the actual service names. A *ServiceStartName* is the user name input on the Start dialog selected from the Service Control Manager.

Debugging services

Debugging service applications can be tricky, because it requires short time intervals:

1. First, launch the application in the debugger. Wait a few seconds until it has finished loading.
2. Quickly start the service from the control panel or from the command line:

   ```
   start MyServ
   ```

You must launch the service quickly (within 15-30 seconds of application startup) because the application will terminate if no service is launched.

Another approach is to attach to the service application process when it is already running. (That is, starting the service first, and then attaching to the debugger). To attach to the service application process, choose Run | Attach To Process, and select the service application in the resulting dialog.

In some cases, this second approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

1. First create a key called **Image File Execution Options** in the following registry location:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
   ```

2. Create a subkey with the same name as your service (for example, MYSERV.EXE). To this subkey, add a value of type REG_SZ, named Debugger. Use the full path to Delphi32.exe as the string value.

3 In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

Creating packages and DLLs

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs. However, on Linux, DLLs (and packages) recompile as shared objects.

Packages are special DLLs used by Delphi applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

The following compiler directives can be placed in library project files:

Table 5.1 Compiler directives for libraries

Compiler Directive	Description
{$LIBPREFIX 'string'}	Adds a specified prefix to the output file name. For example, you could specify {$LIBPREFIX 'dcl'} for a design-time package, or use {$LIBPREFIX ' '} to eliminate the prefix entirely.
{$LIBSUFFIX 'string'}	Adds a specified suffix to the output file name before the extension. For example, use {$LIBSUFFIX '-2.1.3'} in something.pas to generate something-2.1.3.bpl.
{$LIBVERSION 'string'}	Adds a second extension to the output file name after the .bpl extension. For example, use {$LIBVERSION '2.1.3'} in something.pas to generate something.bpl.2.1.3.

For more information on packages, see Chapter 11, "Working with packages and components."

When to use packages and DLLs

For most applications written in Delphi, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

You cannot pass runtime type information (RTTI) across DLLs or from a DLL to an executable. That's because DLLs all maintain their own symbol information. If you need to pass a *TStrings* object from a DLL then using an **is** or **as** operator, you need to create a package rather than a DLL. Packages share symbol information.

Writing database applications

Note Not all versions of Delphi include database support.

One of Delphi's strengths is its support for creating advanced database applications. Delphi supports tools that allow you to connect to SQL servers and databases such as Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, and DB2 while providing transparent data sharing between applications.

Delphi includes many components for accessing databases and representing the information they contain. On the Component palette, the database components are grouped according to the data access mechanism and function.

Table 5.2 Database pages on the Component palette

Palette page	Contents
BDE	Components that use the Borland Database Engine (BDE), a large API for interacting with databases. The BDE supports the broadest range of functions and comes with the most supporting utilities including Database Desktop, Database Explorer, SQL Monitor, and BDE Administrator. See Chapter 20, "Using the Borland Database Engine" for details.
ADO	Components that use ActiveX Data Objects (ADO), developed by Microsoft, to access database information. Many ADO drivers are available for connecting to different database servers. ADO-based components let you integrate your application into an ADO-based environment. See Chapter 21, "Working with ADO components" for details.
dbExpress	Cross-platform components that use dbExpress to access database information. dbExpress drivers provide fast access to databases but need to be used with *TClientDataSet* and *TDataSetProvider* to perform updates. See Chapter 22, "Using unidirectional datasets" for details.
InterBase	Components that access InterBase databases directly, without going through a separate engine layer. For more information about using the InterBase components, see the online Help.
Data Access	Components that can be used with any data access mechanism such as *TClientDataSet* and *TDataSetProvider*. See Chapter 23, "Using client datasets" for information about client datasets. See Chapter 24, "Using provider components"for information about providers.
Data Controls	Data-aware controls that can access information from a data source. See Chapter 15, "Using data controls" for details.

When designing a database application, you must decide which data access mechanism to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

See Part II, "Developing database applications" in this manual for details on how to use Delphi to create both database client applications and application servers. Refer to "Deploying database applications" on page 13-6 for deployment information.

Distributing database applications

Delphi provides support for creating distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including DCOM, CORBA, TCP/IP, and SOAP.

For more information about building distributed database applications, see Chapter 25, "Creating multi-tiered applications."

Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see "Deploying database applications" on page 13-6.

Creating Web server applications

Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests.

You can create several different types of Web server applications using the following Delphi technologies:

- Web Broker
- WebSnap
- InternetExpress
- Web Services

Using Web Broker

You can use Web Broker (also called NetCLX architecture) to create Web server applications such as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Components on the Internet page of the Component palette enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client.

To create a new Web server application using the Web Broker architecture, select File | New | Other and select Web Server Application in the New Items dialog box. Then select the Web server application type:

Table 5.3 Web server applications

Web server application type	Description
ISAPI and NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread. Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.
CGI Stand-alone executable	CGI Web server applications are console applications that receive requests from clients on standard input, process those requests, and sends back the results to the server on standard output to be sent to the client. Selecting this type of application adds the required entries to the **uses** clause of the project file and adds the appropriate $APPTYPE directive to the source.
Win-CGI Stand-alone executable	Win-CGI Web server applications are Windows applications that receive requests from clients from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application. Selecting this type of application adds the required entries to the **uses** clause of the project file and adds the appropriate $APPTYPE directive to the source.
Apache Shared Module (DLL)	Selecting this type of application sets up your project as a DLL. Apache Web server applications are DLLs loaded by the Web server. Information is passed to the DLL, processed, and returned to the client by the Web server.
Web App Debugger Stand-alone executable	Selecting this type of application sets up an environment for developing and testing Web server applications. Web App Debugger applications are executable files loaded by the Web server. This type of application is not intended for deployment.

CGI and Win-CGI applications use more system resources on the server, so complex applications are better created as ISAPI , NSAPI, or Apache DLL applications. If writing cross-platform applications, you should select CGI stand-alone or Apache Shared Module (DLL) for Web server development. These are also the same options you see when creating WebSnap and Web Service applications.

For more information on building Web server applications, see Chapter 27, “Creating Internet applications.”

Creating WebSnap applications

WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other mime content for Web pages. WebSnap is for server side development. WebSnap cannot be used in cross-platform applications at this time.

To create a new WebSnap application, select File | New | Other and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (ISAPI/NSAPI, CGI, Win-CGI, Apache). See Table 5.3, "Web server applications" for details.

For more information on WebSnap, see Chapter 29, "Using WebSnap."

Using InternetExpress

InternetExpress is a set of components that extends the basic Web server application architecture to act as the client of an application server. You use InternetExpress for applications wherein browser-based clients can fetch data from a provider, resolve updates to the provider, while executing on a client.

InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML determines the layout and appearance of the pages displayed in end-user browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in the XML data packets on the client machine.

For more information on InternetExpress, see "Building Web applications using InternetExpress" on page 25-33.

Creating Web Services applications

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language).

Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

You can use Delphi to build servers to implement Web Services and clients that call on those services. You can write clients for arbitrary servers to implement Web Services that respond to SOAP messages, and Delphi servers to publish Web Services for use by arbitrary clients.

Refer to Chapter 31, "Using Web Services" for more information on Web Services.

Writing applications using COM

COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine. You can also use COM+, ActiveX and Active Server Pages.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Using COM and DCOM

Delphi has classes and wizards that make it easy to create COM, OLE, or ActiveX applications. You can create COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms. COM also severs as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories.

Using Delphi to create COM-based applications offers a wide range of possibilities, from improving software design by using interfaces internally in an application, to creating objects that can interact with other COM-based API objects on the system, such as the Win9x Shell extensions and DirectX multimedia support. Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

For more information on COM and Active X controls, see Chapter 33, “Overview of COM technologies,” Chapter 38, “Creating an ActiveX control,” and “Distributing a client application as an ActiveX control” on page 25-32.

For more information on DCOM, see “Using DCOM connections” on page 25-8.

Using MTS and COM+

COM applications can be augmented with special services for managing objects in a large distributed environment. These services include transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) on versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later).

For more information on MTS and COM+, see Chapter 39, “Creating MTS or COM+ objects” and “Using transactional data modules” on page 25-6.

Using data modules

A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are several types of data modules, including standard, remote, Web modules, applet modules, and services, depending on which edition of Delphi you have. Each type of data module serves a special purpose.

- Standard data modules are particularly useful for single- and two-tiered database applications, but can be used to organize the nonvisual components in any application. For more information, see “Creating and editing standard data modules” on page 5-15.
- Remote data modules form the basis of an application server in a multi-tiered database application. They are not available in all editions. In addition to holding the nonvisual components in the application server, remote data modules expose the interface that clients use to communicate with the application server. For more information about using them, see “Adding a remote data module to an application server project” on page 5-19.
- Web modules form the basis of Web server applications. In addition to holding the components that create the content of HTTP response messages, they handle the dispatching of HTTP messages from client applications. See Chapter 27, “Creating Internet applications” for more information about using Web modules.
- Applet modules form the basis of control panel applets. In addition to holding the nonvisual controls that implement the control panel applet, they define the properties that determine how the applet’s icon appears in the control panel and include the events that are called when users execute the applet. For more information about applet modules, see the online Help.
- Services encapsulate individual services in an NT service application. In addition to holding any nonvisual controls used to implement a service, services include the events that are called when the service is started or stopped. For more information about services, see “Service applications” on page 5-4.

Creating and editing standard data modules

To create a standard data module for a project, choose File | New | Data Module. Delphi opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project.

At design time a data module looks like a standard Delphi form with a white background and no alignment grid. As with forms, you can place nonvisual components from the Component palette onto a module, and edit their properties in the Object Inspector. You can resize a data module to accommodate the components you add to it.

You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a data module.

Table 5.4 Context menu options for data modules

Menu item	Purpose
Edit	Displays a context menu with which you can cut, copy, paste, delete, and select the components in the data module.
Position	Aligns nonvisual components to the module's invisible grid (*Align To Grid) or* according to criteria you supply in the Alignment dialog box (*Align*).
Tab Order	Enables you to change the order that the focus jumps from component to component when you press the tab key.
Creation Order	Enables you to change the order that data access components are created at start-up.
Revert to Inherited	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties.
View DFM	Toggles between the formats (binary or text) in which this particular form file is saved.

For more information about data modules, see the online Help.

Naming a data module and its unit file

The title bar of a data module displays the module's name. The default name for a data module is "DataModule*N*" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2." The corresponding unit file for *DataModule2* defaults to "Unit2."

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module:

1 Select the module.

2 Edit the *Name* property for the module in the Object Inspector.

The new name for the module appears in the title bar when the *Name* property in the Object Inspector no longer has focus.

Changing the name of a data module at design time changes its variable name in the interface section of code. It also changes any use of the type name in procedure declarations. You must manually change any references to the data module in code you write.

To rename a unit file for a data module:

1 Select the unit file.

Placing and naming components

You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate page of the Component palette, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, Delphi assigns it a generic name that identifies what kind of component it is, followed by a *1*. For example, the *TDataSource* component adopts the name *DataSource1*. This makes it easy to select specific components whose properties and methods you want to work with.

You may still want to name a component a different name that reflects the type of component and what it is used for.

To change the name of a component in a data module:

1 Select the component.
2 Edit the component's *Name* property in the Object Inspector.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the Object Inspector no longer has focus.

For example, suppose your database application uses the CUSTOMER table. To access the table, you need a minimum of two data access components: a data source component (*TDataSource)* and a table component (*TClientDataSet)*. When you place these components in your data module, Delphi assigns them the names *DataSource1* and *ClientDataSet1*. To reflect the type of component and the database they access, CUSTOMER, you could change these names to *CustomerSource* and *CustomerTabl*e.

Using component properties and events in a data module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to *True* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1*, to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms. The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a *TDataSource* component has three possible events: *OnDataChang*e, *OnStateChang*e, and *OnUpdateDat*a. A *TClientDataSet* component has over 20 potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

Creating business rules in a data module

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module. The prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:

```
type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
    Orders: TClientDataSet;
    :
  private
    { Private declarations }
  public
    { Public declarations }
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;

var
  CustomerData: TCustomerData;
```

The procedures and functions you write should follow in the implementation section of the code for the module.

Accessing a data module from a form

To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways:

- In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section.
- Click the form's unit file, choose File | Use Unit, and enter the name of the module or pick it from the list box in the Use Unit dialog.
- For database components, in the data module click a dataset or query component to open the Fields editor and drag any existing fields from the editor onto the form. Delphi prompts you to confirm that you want to add the module to the form's **uses** clause, then creates controls (such as edit boxes) for the fields.

For example, if you've added the *TClientDataSet* component to your data module, double-click it to open the Fields editor. Select a field and drag it to the form. An edit box component appears.

Because the data source is not yet defined, Delphi adds a new data source component, *DataSource1,* to the form and sets the edit box's *DataSource* property to *DataSource1*. The data source automatically sets its *DataSet* property to the dataset component, *ClientDataSet1,* in the data module.

You can define the data source *before* you drag a field to the form by adding a *TDataSource* component to the data module. Set the data source's *DataSet* property to *ClientDataSet1.* After you drag a field to the form, the edit box appears with its *TDataSource* property already set to *DataSource1*. This method keeps your data access model cleaner.

Adding a remote data module to an application server project

Some editions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.

To add a remote data module to a project:

1. Choose File | New | Other.
2. Select the Multitier page in the New Items dialog box.
3. Double-click the desired type of module (CORBA Data Module, Remote Data Module, or Transactional Data Module) to open the Remote Data Module wizard.

Once you add a remote data module to a project, you use it just like a standard data module.

For more information about multi-tiered database applications, see Chapter 25, "Creating multi-tiered applications."

Using the Object Repository

The Object Repository (Tools | Repository) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The repository is maintained in DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

Sharing items within a project

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (File | New | Other), you'll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

Adding items to the Object Repository

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository. To add an item to the Object Repository,

1. If the item is a project or is in a project, open the project.

2 For a project, choose Project | Add To Repository. For a form or data module, right-click the item and choose Add To Repository.

3 Type a description, title, and author.

4 Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, Delphi creates a new page.

5 Choose Browse to select an icon to represent the object in the Object Repository.

6 Choose OK.

Sharing objects in a team environment

You can share objects with your workgroup or development team by making a repository available over a network. To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

1 Choose Tools | Environment Options.

2 On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the repository, Delphi creates a DELPHI32.DRO file in the Shared Repository directory if one doesn't exist already.

Using an Object Repository item in a project

To access items in the Object Repository, choose File | New | Other. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- Copy
- Inherit
- Use

Copying an item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

Inheriting an item

Choose Inherit to derive a new class from the selected item in the Object Repository and add the new class to your project. When you recompile your project, any changes that have been made to the item in the Object Repository will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

Using an item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

Using project templates

Templates are predesigned projects that you can use as starting points for your own work. To create a new project from a template,

1. Choose File | New | Other to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog, specify a directory for the new project's files.

Delphi copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

Modifying shared items

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with other developers.

Specifying a default project, new form, and main form

By default, when you choose File | New | Application or File | New | Form, Delphi displays a blank form. You can change this behavior by reconfiguring the Repository:

1. Choose Tools | Repository
2. If you want to specify a default project, select the Projects page and choose an item under Objects. Then select the New Project check box.

3 If you want to specify a default form, select a Repository page (such as Forms), them choose a form under Objects. To specify the default new form (File | New | Form), select the New Form check box. To specify the default main form for new projects, select the Main Form check box.

4 Click OK.

Enabling Help in applications

Both the VCL and CLX support displaying Help from applications using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers. To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager.

The VCL provides to all applications an instance of *TWinHelpViewer*, which implements all of these interfaces and provides a link between applications and WinHelp; CLX requires that application developers provide their own implementation.

The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer if it can provide support for a particular Help keyword or context, and then it passes the Help request on to the viewer which says it can provide such support. (If more than one viewer supports the keyword, as would be the case in an application which had registered viewers for both Man and Info, the Help Manager can display a selection box through which the user of the application can determine which Help viewer to invoke. Otherwise, it displays the first responding Help system encountered).

Help system interfaces

The Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in HelpIntfs.pas, which also contains the implementation of the Help Manager.

ICustomHelpViewer provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer.

IExtendedHelpViewer provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, “IntToStr” might be a keyword in the Help system, but “String manipulation routines” could be the name of a topic).

*ISpecialWinHelpViewe*r provides support for responding to specialized WinHelp messages that an application running under Windows may receive and which are not easily generalizable. In general, only applications operating in the Windows environment need to implement this interface, and even then it is only required for applications that make extensive use of non-standard WinHelp messages.

IHelpManager provides a mechanism for the Help viewer to communicate back to the application's Help Manager and request additional information. An *IHelpManager* is obtained at the time the Help viewer registers itself.

IHelpSystem provides a mechanism through which *TApplication* passes Help requests on to the Help system. *TApplication* obtains an instance of an object which implements both *IHelpSystem* and *IHelpManager* at application load time and exports that instance as a property; this allows other code within the application to file Help requests directly when appropriate.

IHelpSelector provides a mechanism through which the Help system can invoke the user interface to ask which Help viewer should be used in cases where more than one viewer is capable of handling a Help request, and to display a Table of Contents. This display capability is not built into the Help Manager directly to allow the Help Manager code to be identical regardless of which widget set or class library is in use.

Implementing ICustomHelpViewer

The *ICustomHelpViewer* interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents.

Communicating with the Help Manager

ICustomHelpViewer provides four functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

The Help Manager calls through these functions in the following circumstances:

- *ICustomHelpViewer.GetViewerName : String* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* is called **immediately** following registration to provide the viewer with a unique cookie that identifies it. This information must be stored off for later use; if the viewer shuts down on its own (as opposed to in response to a notification from the Help Manager), it must provide the Help Manager with the identifying cookie so that the Help Manager can release all references to the viewer. (Failing to provide the cookie, or providing the wrong one, causes the Help Manager to potentially release references to the wrong viewer.)

- *ICustomHelpViewer.ShutDown* is called by the Help Manager to notify the Help viewer that the Manager is shutting down and that any resources the Help viewer has allocated should be freed. It is recommended that all resource freeing be delegated to this method.
- *ICustomHelpViewer.SoftShutDown* is called by the Help Manager to ask the Help viewer to close any externally visible manifestations of the help system (for example, windows displaying help information) without unloading the viewer.

Asking the Help Manager for information

Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things: a request for the window handle of the currently active control; a request for the name of the Help file which the Help Manager believes should contain help for the currently active control; a request for the path to that Help file; and a notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that it do so.

IHelpManager.GetHandle : LongInt is called by the Help viewer if it needs to know the handle of the currently active control; the result is a window handle.

IHelpManager.GetHelpFile: String is called by the Help viewer if it wishes to know the name of the Help file which the currently active control believes contains its help.

IHelpManager.Release is called to notify the Help Manager when a Help viewer is disconnecting. It should *never* be called in response to a request through *ICustomHelpViewer.ShutDown*; it is only used to notify the Help Manager of unexpected disconnects.

Displaying keyword-based Help

Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier. (Numeric help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system; while CLX supports them, they are not recommended for use in CLX applications because most Linux Help systems do not understand them.) *ICustomHelpViewer* implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests.

ICustomHelpViewer provides three methods for handling keyword-based Help:

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.UnderstandsKeyword(const HelpString: String): Integer
```

is the first of the three methods called by the Help Manager, which will call *each* registered Help viewer with the same string to ask if the viewer provides help for that string; the viewer is expected to respond with an integer indicating how many different Help pages it can display in response to that Help request. The viewer can use any method it wants to determine this — inside the IDE, the HyperHelp viewer maintains its own index and searches it. If the viewer does not support help on this keyword, it should return zero. Negative numbers are currently interpreted as meaning zero, but this behavior is not guaranteed in future releases.

```
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
```

is called by the Help Manager if more than one viewer can provide help on a topic. The viewer is expected to return a *TStringList*. The strings in the returned list should map to the pages available for that keyword, but the characteristics of that mapping can be determined by the viewer. In the case of the HyperHelp viewer, the string list always contains exactly one entry (HyperHelp provides its own indexing, and duplicating that elsewhere would be pointless duplication); in the case of the Man page viewer, the string list consists of multiple strings, one for each section of the manual which contains a page for that keyword.

```
ICustomHelpViewer.ShowHelp(const HelpString: String)
```

is called by the Help Manager if it needs the Help viewer to display help for a particular keyword. This is the last method call in the operation; it is guaranteed to never be called unless *CanShowKeyword* is invoked first.

Displaying tables of contents

ICustomHelpViewer provides two methods relating to displaying tables of contents:

- *CanShowTableOfContents*
- *ShowTableOfContents*

The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling *ICustomHelpViewer.CanShowTableOfContents : Boolean* and then invokes a particular Help viewer by calling *ICustomHelpViewer.ShowTableOfContents*.

It is reasonable for a particular viewer to refuse to allow requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports a table of contents, on the other hand, by passing the request to display a table of contents directly to HyperHelp. It is *not* reasonable, however, for an implementation of *ICustomHelpViewer* to respond to queries through *CanShowTableOfContents* with the answer *true*, and then ignore requests through *ShowTableOfContents*.

Implementing IExtendedHelpViewer

ICustomHelpViewer only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself.

Applications written in CLX can talk to systems requiring context-based Help by extending the object which implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking to Help systems that allow you to jump directly to high-level topics instead of using keyword searches.

IExtendedHelpViewer exposes four functions. Two of them — *UnderstandsContext* and *DisplayHelpByContext* — are used to support context-based Help; the other two — *UnderstandsTopic* and *DisplayTopic* — are used to support topics.

When an application user presses F1, the Help Manager calls

```
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
```

and the currently activated control supports context-based, rather than keyword-based Help. As with *ICustomHelpViewer.CanShowKeyword*, the Help Manager queries all registered Help viewers iteratively. Unlike the case with *ICustomHelpViewer.CanShowKeyword*, however, if more than one viewer supports a specified context, the *first* registered viewer with support for a given context is invoked.

The Help Manager calls

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
```

after it has polled the registered Help viewers.

The topic support functions work the same way:

```
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
```

is used to poll the Help viewers asking if they support a topic;

```
IExtendedHelpViewer.DisplayTopic(const Topic: String)
```

is used to invoke the first registered viewer which reports that it is able to provide help for that topic.

Implementing IHelpSelector

IHelpSelector is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application.

To override the decision of the Help Manager in such cases, an application must register a class that provides an implementation of the *IHelpSelector* interface. *IHelpSelector* exports two functions: *SelectKeyword*, and *TableOfContents*. Both take as arguments a *TStrings* containing, one by one, either the possible keyword matches or the names of the viewers claiming to provide a table of contents. The implementor is required to return the index (in the *TStrings*) that represents the selected string.

Note The Help Manager may get confused if the strings are re-arranged; it is recommended that implementors of *IHelpSelector* refrain from doing this. The Help system only supports *one* HelpSelector; when new selectors are registered, any previously existing selectors are disconnected.

Registering Help system objects

For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer*, *IExtendedHelpViewer*, *ISpecialWinHelpViewer*, and *IHelpSelector* must register with the Help Manager.

To register Help system objects with the Help Manager, you need to

- Register the Help viewer
- Register the Help Selector

Registering Help viewers

The unit that contains the object implementation must use HelpIntfs. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must assign the instance variable and pass it to the function *RegisterViewer*. *RegisterViewer* is a flat function exported by HelpIntfs.pas which takes as an argument an *ICustomHelpViewer* and returns an *IHelpManager*. The *IHelpManager* should be stored for future use.

Registering Help selectors

The unit that contains the object implementation must use HelpIntfs and QForms. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must register the Help selector through the *HelpSystem* property of the global Application object:

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

This procedure does not return a value.

Using Help in a VCL Application

The following sections explain how to use Help within a VCL application.

- How TApplication processes VCL Help
- How VCL controls process Help
- Calling a Help system directly
- Using IHelpSystem

How TApplication processes VCL Help

TApplication in the VCL provides four methods that are accessible from application code:

Table 5.5 Help methods in TApplication

HelpCommand	Takes a Windows Help style HELP_COMMAND and passes it off to WinHelp. Help requests forwarded through this mechanism are passed only to implementations of IspecialWinHelpViewer.
HelpContext	Invokes the Help System with a request for context-based Help.
HelpKeyword	Invokes the HelpSystem with a request for keyword-based Help.
HelpJump	Requests the display of a particular topic.

All four functions take the data passed to them and forward it through a data member of *TApplication* which represents the Help System. That data member is directly accessible through the property *HelpSystem*.

How VCL controls process Help

All controls that derive from *TControl* expose three properties which are used by the Help system: *HelpSystem*, *HelpType*, *HelpContext*, and *HelpKeyword*.

The *HelpType* property contains an instance of an enumerated type that determines if the control's designer expects help to be provided via keyword-based Help or context-based Help. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, that can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of Help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWinControl* calls *InvokeHelp*.

Using Help in a CLX Application

The following sections explain how to use Help within a CLX application.

- How TApplication processes CLX Help
- How CLX controls process Help
- Calling a Help system directly
- Using IHelpSystem

How TApplication processes CLX Help

TApplication in CLX provides two methods that are accessible from application code:

- *ContextHelp*, which invokes the Help system with a request for context-based Help
- *KeywordHelp*, which invokes the Help system with a request for keyword-based Help

Both functions take as an argument the context or keyword being passed and forward the request on through a data member of *TApplication,* which represents the Help system. That data member is directly accessible through the read-only property *HelpSystem*.

How CLX controls process Help

All controls that derive from *TControl* expose four properties which are used by the Help system: *HelpType*, *HelpFile*, *HelpContext*, and *HelpKeyword*. *HelpFile* is supposed to contain the name of the file in which the control's help is located; if the help is located in an external Help system that does not care about file names (say, for example, the Man page system), then the property should be left blank.

The *HelpType* property contains an instance of an enumerated type which determines if the control's designer expects help to be provided via keyword-based Help or context-based Help; the other two properties are linked to it. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, which can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWidgetControl* calls *InvokeHelp*.

Calling a Help system directly

For additional Help system functionality not provided by the VCL or CLX, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.

Using IHelpSystem

IHelpSystem allows a VCL or CLX application to do three things:

- Provides path information to the Help Manager
- Provides a new Help selector
- Asks the Help Manager to display help

Assigning a Help selector allows the Help Manager to delegate decision-making in cases where multiple external Help systems can provide help for the same keyword. For more information, see the section “Implementing IHelpSelector” on page 5-26.

IHelpSystem exports four procedures and one function to request the Help Manager to display help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

Hook is intended entirely for WinHelp compatibility and should not be used in a CLX application; it allows processing of WM_HELP messages that cannot be mapped directly onto requests for keyword-based, context-based, or topic-based Help. The other methods each take two arguments: the keyword, context ID, or topic for which help is being requested, and the Help file in which it is expected that help can be found.

In general, unless you are asking for topic-based help, it is equally effective and more clear to pass help requests to the Help Manager through the *InvokeHelp* method of your control.

Customizing the IDE Help system

The Delphi IDE supports multiple Help viewers in exactly the same way that a VCL or CLX application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same WinHelpViewer that the VCL uses.

To install a new Help viewer in the IDE, you do exactly what you would do in a CLX application, with one difference. You write an object that implements *ICustomHelpViewer* (and, if desired, *IExtendedHelpViewer*) to forward Help requests to the external viewer of your choice, and you register the *ICustomHelpViewer* with the IDE.

To register a custom Help viewer with the IDE,

1. Make sure that the unit implementing the Help viewer contains HelpIntfs.pas.
2. Build the unit into a design-time package registered with the IDE, and build the package with runtime packages turned on. (This is necessary to ensure that the Help Manager instance used by the unit is the same as the Help Manager instance used by the IDE.)
3. Make sure that the Help viewer exists as a global instance within the unit.
4. In the initialization section of the unit, make sure that the instance is passed to the *RegisterHelpViewer* function.

Chapter

6

Developing the application user interface

With Delphi, you design a user interface (UI) by selecting components from the component palette and dropping them onto forms. You get the components to do what you want by setting their properties and coding their event handlers.

Controlling application behavior

TApplication, *TScreen*, and *TForm* are the classes that form the backbone of all Delphi applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information such as screen resolution and available display fonts. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

Using the main form

TForm is the key class for creating GUI applications. When you open Delphi displaying a default project or when you create a new project, a form is displayed on which you can begin your UI design.

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form,

1 Choose Project | Options and select the Forms page.

2 In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed.

Adding forms

To add a form to your project, select File | New Form. You can see all your project's forms and their associated units listed in the Project Manager (View | Project Manager) and you can display a list of the forms alone by choosing View | Forms.

Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form,

1 Select the form that needs to refer to another.
2 Choose File | Use Unit.
3 Select the name of the form unit for the form to be referenced.
4 Choose OK.

Linking a form to another just means that the **uses** clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the File | Use Unit command does.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both **uses** clauses in the **interface** parts of their respective unit files. This will generate the "Circular reference" error at compile time.

Hiding the main form

You can prevent the main form from displaying when your application first starts up. To do so, you must use the global *Application* variable (described in the next topic).

To hide the main form at startup,

1 Choose Project | View Source to display the main project file.

2 Add the following lines after the call to `Application.CreateForm` and before the call to `Application.Run`.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

Note You can set the form's *Visible* property to *False* using the Object Inspector at design time rather than setting it at runtime as shown above.

Working at the application level

The global variable *Application*, of type *TApplication*, is in every VCL or CLX based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* would handle how you would call a help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the Project | Options Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnMessage* event lets you intercept Windows messages (on Windows only), the *OnEvent* event lets you intercept events, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

Handling the screen

A global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying

- the look of the cursor
- the size of the window in which your application is running
- a list of fonts available to the screen device
- multiple screen behavior (not available for cross-platform)

If your Windows application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.

If using CLX for cross-platform programming, the default behavior is that applications create a screen component based on information about the current screen device and assign it to Screen.

Managing layout

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

TControl introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
MaxHeight: Integer) of object;
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as `var` parameters which can be changed inside the event handler. *OnConstrainedResize* is published for container objects (*TForm*, *TScrollBox*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

Responding to event notification

The operating system will notify your application when an event has occurred (such as a mouse click, keystrokes entered, and so on) while it is running. The underlying way that event notifications are handled by VCL and CLX objects is different, but the way you work with event notifications at the component level is typically the same. Components have events and methods built-in for the most commonly occurring events. You can use the methods provided with the component in most cases. If you need to write additional event handling, you can override an existing method to write your own. Unless you are writing your own components, you do not need to change the underlying event notification schema.

VCL If developing applications for Windows only, you need to understand that Windows is a message-based operating system. System messages are handled by a message handler that translates the message to an event or event handler. The message itself is a record passed to a control by Windows. For instance, when you click a mouse button on a dialog box, Windows sends a message to the active control and the application containing that control reacts to this new event. If the click occurs over a button, the *OnClick* event could be activated upon receipt of the message. If the click occurs just in the form, the application can ignore the message.

The record type passed to the application by Windows is called a *TMsg*. Windows predefines a constant for each message, and these values are stored in the message field of the *TMsg* record. Each of these constants begin with the letters wm.

The VCL automatically handles messages unless you override the message handling system and create your own message handlers. For more information on messages and message handling, see “Understanding the message-handling system” on page 46-1, “Changing message handling” on page 46-3, and “Creating new message handlers” on page 46-5.

CLX For cross-platform programming: The operating system notification that an event occurred is sent to the underlying Qt widget layer where it is translated into an event and eventually into event objects by *HookEvents*. *EventFilter* is called automatically when a CLX control needs to handle a Qt mouse or keyboard event.

EventFilter responds to event notifications by performing the default response. Typically, this involves dispatching the event to the appropriate virtual method (such as the *Click* method, which generates an *OnClick* event).

CLX Note When overriding the *EventFilter* method, you need to call the inherited method so that the default event processing can occur.

Using forms

When you create a form in Delphi from the IDE, Delphi automatically creates the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don’t have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default Delphi behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

Controlling when forms reside in memory

By default, Delphi automatically creates the application's main form in memory by including the following code in the application's main entry point:

```
Application.CreateForm(TForm1, Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form's class and is used to reference the form while the application is running. Any unit that includes the form's unit in its **uses** clause can access the form via this variable.

All forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

Displaying an auto-created form

If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm.ShowModal;
end;
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

Creating forms dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE, you:

1 Select the File | New Form from the main menu to display the new form.

2 Remove the form from the Auto-create forms list of the Project | Options | Forms page.

This removes the form's invocation. As an alternative, you can manually remove the following line from program's main entry point:

```
Application.CreateForm(TResultsForm, ResultsForm);
```

3 Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
ResultsForm:=TResultForm.Create(self);
try
  ResultsForm.ShowModal;
finally
  ResultsForm.Free;
end;
```

In the above example, note the use of **try..finally**. Putting in the line `ResultsForm.Free;` in the **finally** clause ensures that the memory for the form is freed even if the form raises an exception.

The event handler in the example deletes the form after it is closed, so the form would need to be recreated if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

Note If you create a form using its constructor, be sure to check that the form is not in the Auto-create forms list on the Project Options | Forms page. Specifically, if you create the new form without deleting the form of the same name from the list, Delphi creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global variable no longer points to a valid form. Any attempt to use the global variable will likely crash the application.

Creating modeless forms such as windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

Using a local variable to create a form instance

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

Passing additional arguments to forms

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be **nil**.

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

```
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  inherited Create(Owner);
  case whichButton of
```

```
      1: ResultsLabel.Caption := 'You picked the first button.';
      2: ResultsLabel.Caption := 'You picked the second button.';
      3: ResultsLabel.Caption := 'You picked the third button.';
    end;
  end;
```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;
```

Retrieving data from forms

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

Retrieving data from modeless forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red", "Green", "Blue", and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor*, to store the actual value for the property in the private data member *FColor*:

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton)* on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {do something with the string MainColor}
  end;
end;
```

The event handler first verifies that *ColorForm* exists using the *Assigned* function. It then gets the value of *ColorForm's CurrentColor* property.

Alternatively, if *ColorForm* had a public function named *GetColor*, another form could get the current color without using the *CurrentColor* property (for example, `MainColor := ColorForm.GetColor;`). In fact, there's nothing to prevent another form from getting the *ColorForm's* currently selected color by checking the listbox selection directly:

```
with ColorForm.ColorListBox do
  MainColor := Items[ItemIndex];
```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

Retrieving data from modal forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* argument. Presumably, this *Pointer* points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

The constructor saves the pointer to a private data member *FColor* and initializes the string to an empty string.

Note To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see "Controlling when forms reside in memory" on page 6-6.

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

Notice that the event handler stores the selected color name in the string referenced by the pointer that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked.

The event handler would look as follows:

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;

procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

UpdateButtonClick creates a String called MainColor. The address of MainColor is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to MainColor as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in MainColor, assuming that a color was selected. Otherwise, MainColor contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having MainColor default to an empty string).

Reusing components and groups of components

Delphi offers several ways to save and reuse work you've done with components:

- *Component templates* provide a simple, quick way of configuring and saving groups of components. See "Creating and using component templates" on page 6-13.
- You can save forms, data modules, and projects in the *Repository*. This gives you a central database of reusable elements and lets you use form inheritance to propagate changes. See "Using the Object Repository" on page 5-19.
- You can save *frames* on the component palette or in the repository. Frames use form inheritance and can be embedded into forms or other frames. See "Working with frames" on page 6-13.
- Creating a *custom component* is the most complicated way of reusing code, but it offers the greatest flexibility. See Chapter 40, "Overview of component creation."

Creating and using component templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the component palette, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template,

1. Place and arrange components on a form. In the Object Inspector, set their properties and events as desired.
2. Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.
3. Choose Component | Create Component Template.
4. Specify a name for the component template in the Component Name edit box. The default proposal is the component type of the first component selected in step 2 followed by the word “Template”. For example, if you select a label and then an edit box, the proposed name will be “TLabelTemplate”. You can change this name, but be careful not to duplicate existing component names.
5. In the Palette Page edit box, specify the component palette page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.
6. Under Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.
7. Click OK.

To remove templates from the component palette, choose Component | Configure Palette.

Working with frames

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties.

In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the component palette for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Frames are useful to organize groups of controls that are used in multiple places in your application. For example, if you have a bitmap that is used on multiple forms, you can put it in a frame and only one copy of that bitmap is included in the resources of your application. You could also describe a set of edit fields that are intended to edit a table with a frame and use that whenever you want to enter data into the table.

Creating frames

To create an empty frame, choose File | New | Frame, or choose File | New and double-click on Frame. You can then drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose File | New | Application, close the new form and unit without saving them, then choose File | New | Frame and save the project.

Note When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing View | Forms and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame’s form file by right-clicking and choosing View as Form or View as Text.

Adding frames to the component palette

Frames are added to the component palette as component templates. To add a frame to the component palette, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click on the frame, and choose Add to Palette. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

Using and modifying frames

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

- Select a frame from the component palette and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame’s unit file in your project.

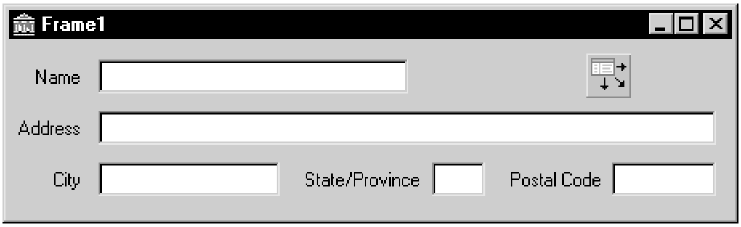
- Select *Frames* from the Standard page of the component palette and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Delphi declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Delphi declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed.

If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

Figure 6.1 A frame with data-aware controls and a data source component



In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the *Picture* property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Imag*e object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *TImage.Picture* once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

Sharing frames

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.pas) and form (.dfm or .xfm) files.

To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository For more information, see “Using the Object Repository” on page 5-19.

If you send a frame’s unit and form files to other developers, they can open them and add them to the component palette. If the frame has other frames embedded in it, they will have to open it as part of a project.

Organizing actions for toolbars and menus

Delphi provides several features that simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon.

Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.

On the Windows platform, tools are provided to make it easy to define and group actions, create different layouts, and customize menus at design time or runtime. These tools are known collectively as ActionBand tools, and the menus and toolbars you create with them are known as action bands. In general, you can create an ActionBand user interface as follows:

- Build the action list to create a set of actions that will be available for your application (use the Action Manager, *TActionManager*)
- Add the user interface elements to the application (use ActionBand components such as *TActionMainMenuBar* and *TActionToolBar*)
- Drag and drop actions from the Action Manager onto the user interface elements

The following table defines the terminology related to setting up menus and toolbars:

Table 6.1 Action setup terminology

Term	Definition
Action	A response to something a user does, such as clicking a menu item. Many standard actions that are frequently required are provided for you to use in your applications as is. For example, file operations such as File Open, File Save As, File Run, and File Exit are included along with many others for editing, formatting, searches, help, dialogs, and window actions. You can also program custom actions and access them using action lists and the Action Manager.
Action band	A container for a set of actions associated with a customizable menu or toolbar. The ActionBand components for main menus and toolbars (*TActionMainMenuBar* and *TActionToolBar*) are examples of action bands.
Action category	Lets you group actions and drop them as a group onto a menu or toolbar. For example, one of the standard action categories is Search which includes Find, FindFirst, FindNext, and Replace actions all at once.

Table 6.1 Action setup terminology (continued)

Term	Definition
Action classes	Classes that perform the actions used in your application. All of the standard actions are defined in action classes such as *TEditCopy*, *TEditCut*, and *TEditUndo*. You can use these classes by dragging and dropping them from the Customize dialog onto an action band.
Action client	Most often represents a menu item or a button that receives a notification to initiate an action. When the client receives a user command (such as a mouse click), it initiates an associated action.
Action list	Maintains a list of actions that your application can take in response to something a user does.
Action Manager	Groups and organizes logical sets of actions that can be reused on ActionBand components. See *TActionManager*.
Menu	Lists commands that the user of the application can execute by clicking on them. You can create menus by using the ActionBand menu class *TActionMainMenuBar*, or by using cross-platform components such as *TMainMenu* or *TPopupMenu*.
Target	Represents the item an action does something to. The target is usually a control, such as a memo or a data control. Not all actions require a target. For example, the standard help actions ignore the target and simply launch the help system.
Toolbar	Displays a visible row of button icons which, when clicked, cause the program to perform some action, such as printing the current document. You can create toolbars by using the ActionBand toolbar component *TActionToolBar*, or by using the cross-platform component *TToolBar*.

If you are doing cross-platform development, refer to “Using action lists” on page 6-23.

What is an action?

As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, Tools|Customize).

An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond when a control fires, for example, when the application user clicks a button or chooses a menu item. You can create a repertoire of actions that are available to your application through menus, through buttons, through toolbars, context menus, and so on.

Actions are associated with other components:

- **Clients:** One or more clients use the action. The client most often represents a menu item or a button (for example, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton*, and so on). Actions also reside on ActionBand components such as *TActionMainMenuBar* and *TActionToolBar*. When the client receives a user command (such as a mouse click), it initiates an associated action. Typically, a client’s *OnClick* event is associated with its action’s *Execute* event.

- **Target:** The action acts on the target. The target is usually a control, such as a memo or a data control. Component writers can create actions specific to the needs of the controls they design and use, and then package those units to create more modular applications. Not all actions use a target. For example, the standard help actions ignore the target and simply launch the help system.

  A target can also be a component. For example, data controls change the target to an associated dataset.

The client influences the action—the action responds when a client fires the action. The action also influences the client—action properties dynamically update the client properties. For example, if at runtime an action is disabled (by setting its *Enabled* property to *False*), every client of that action is disabled, appearing grayed.

You can add, delete, and rearrange actions using the Action Manager or the Action List editor (displayed by double-clicking an action list object, *TActionList*). These actions are later connected to client controls.

Setting up action bands

Because actions do not maintain any “layout” (either appearance or positional) information, Delphi provides action bands which are capable of storing this data. Action bands provide a mechanism that allows you to specify layout information and a set of controls. You can render actions as UI elements such as toolbars and menus.

You organize sets of actions using the Action Manager (*TActionManager*). You can use standard actions provided or create new actions of your own.

You then create the action bands:

- Use *TActionMainMenuBar* to create a main menu.
- Use *TActionToolBar* to create a toolbar.

The action bands act as containers that hold and render sets of actions. You can drag and drop items from the Action Manager editor onto the action band at design time. At runtime, application users can also customize the application’s menus or toolbars using a dialog box similar to the Action Manager editor.

Creating toolbars and menus

Note This section describes the recommended method for creating menus and toolbars in Windows applications. For cross-platform development, you need to use *TToolBar* and the menu components, such as *TMainMenu,* organizing them using action lists (*TActionList*). See “Setting up action lists” on page 6-23.

You use the Action Manager to automatically generate toolbars and main menus based on the actions contained in your application. The Action Manager manages standard actions and any custom actions that you have written. You then create UI elements based on these actions and use action bands to render the actions items as either menu items or as buttons on a toolbar.

The general procedure for creating menus, toolbars, and other action bands involves these steps:

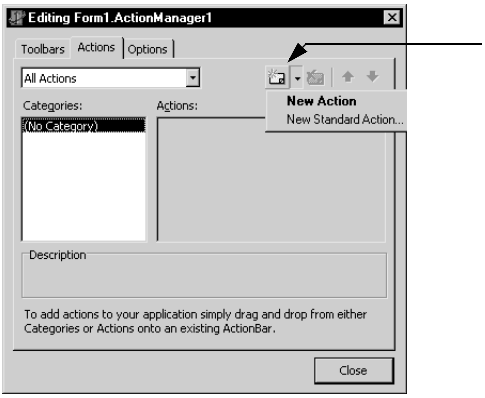
- Drop an Action Manager onto a form.
- Add actions to the Action Manager, which organizes them into appropriate action lists.
- Create the action bands (that is, the menu or the toolbar) for the user interface.
- Drag and drop the actions into the application interface.

The following procedure explains these steps in more detail.

To create menus and toolbars using action bands:

1. From the Additional page of the component palette, drop an Action Manager component (*TActionManager*) onto the form where you want to create the toolbar or menu.
2. If you want images on the menu or toolbar, drop an ImageList component from the Win32 page of the component palette onto a form. (You need to add the images you want to use to the ImageList or use the one provided.)
3. From the Additional page of the component palette, drop one or more of the following action bands onto the form:
   - *TActionMainMenuBar* (for designing main menus)
   - *TActionToolBar* (for designing toolbars)
4. Connect the ImageList to the Action Manager: with focus on the Action Manager and in the Object Inspector, select the name of the ImageList from the Images property.
5. Add actions to the Action Manager editor's action pane:
   - Double-click the Action Manager to display the Action Manager editor.
   - Click the drop-down arrow next to the New Action button (the leftmost button at the top right corner of the Actions tab, as shown in Figure 6.2) and select "New Action..." or "New Standard Action...". A tree view is displayed. Add one or more actions or categories of actions to the Action Manager's actions pane. The Action Manager adds the actions to its action lists.

Figure 6.2 The Action Manager editor.



To add actions to your application simply drag and drop from either Categories or Actions onto an existing ActionBar.

New Action button & dropdown button

6 Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar you are designing.

To add user-defined actions, create a new *TAction* by pressing the New Action button and writing an event handler that defines how it will respond when fired. See “What happens when an action fires” on page 6-24 for details. Once you’ve defined the actions, you can drag and drop them onto menus or toolbars like the standard actions.

Adding color, patterns, or pictures to menus, buttons, and toolbars

You can use the *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on a menu item or button. These properties also let you set up a banner the runs up the left or right side of a menu.

You assign backgrounds and layouts to subitems from their action client objects. If you want to set the background of the items in a menu, in the form designer click on the menu item that contains the items. For example, selecting File lets you change the background of items appearing on the File menu. You can assign a color, pattern, or bitmap in the *Background* property in the Object Inspector.

Use the *BackgroundLayout* property to describe how to place the background on the element. Colors or images can be placed behind the caption normally, stretched to fit the item area, or tiled in small squares to cover the area.

Items with normal (blNormal), stretched (blStretch), or tiled (blTile) backgrounds are rendered with a transparent background. If you create a banner, the full image is placed on the left (blLeftBanner) or the right (blRightBanner) of the item. You need to make sure it is the correct size because it is not stretched or shrunk to fit.

To change the background of an action band (that is, on a main menu or toolbar), select the action band and choose the *TActionClientBar* through the action band collection editor. You can set *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on the entire toolbar or menu.

Adding icons to menus and toolbars

You can add icons next to menu items or replace captions on toolbars with icons. You organize bitmaps or icons using an ImageList.

1. Drop an ImageList component from the Win32 page of the component palette onto a form.
2. Add the images you want to use to the ImageList: Double-click the ImageList. Click Add and navigate to the images you want to use and click OK when done. Some sample images are included in Program Files\Common Files\Borland Shared\Images. (The buttons images include two views of each for active and inactive buttons.)
3. From the Additional page of the component palette, drop one or more of the following action bands onto the form:
   - *TActionMainMenuBar* (for designing main menus)
   - *TActionToolBar* (for designing toolbars)
4. Connect the ImageList to the Action Manager. First, set the focus on the Action Manager. Next, in the Object Inspector, select the name of the ImageList from the Images property.
5. Use the Action Manager editor to add actions to the Action Manager. You can associate an image with an action by setting its *ImageIndex* property to its number in the ImageList.
6. Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar.
7. For toolbars where you only want to display the icon and no caption: select the Toolbar action band and double-click its Items property. In the collection editor, you can select one or more items and set their Caption properties.
8. The images automatically appear on the menu or toolbar.

Creating toolbars and menus that users can customize

You can use action bands with the Action Manager to create customizable toolbars and menus. At runtime, users of your application can customize the toolbars and menus (action bands) in the application user interface using a customization dialog similar to the Action Manager editor.

To allow the user of your application to customize an action band in your application:

1. Drop an Action Manager component onto a form.
2. Drop your action band components (*TActionMainMenuBar*, *TActionToolBar*).
3. Double-click the Action Manager to display the Action Manager editor:
   - Add the actions you want to use in your application. Also add the Customize action, which appears at the bottom of the standard actions list.

- Drop a *TCustomizeDlg* component from the Dialogs tab onto the form, and connect it to the Action Manager using its ActionManager property. You specify a filename for where to stream customizations made by users.
- Drag and drop the actions onto the action band components. (Make sure you add the Customize action to the toolbar or menu.)

4 Complete your application.

When you compile and run the application, users can access a Customize command that displays a customization dialog box similar to the Action Manager editor. They can drag and drop menu items and create toolbars using the same actions you supplied in the Action Manager.

Hiding unused items and categories in action bands

One benefit of using ActionBands is that unused items and categories can be hidden from the user. Over time, the action bands become customized for the application users, showing only the items that they use and hiding the rest from view. Hidden items can become visible again when the user presses a dropdown button. Also, the user can restore the visibility of all action band items by resetting the usage statistics from the customization dialog. Item hiding is the default behavior of action bands, but that behavior can be changed to prevent hiding of individual items, all the items in a particular collection (like the File menu), or all of the items in a given action band.

The action manager keeps track of the number of times an action has been called by the user, which is stored in the associated *TActionClientItem*'s *UsageCount* field. The action manager also records the number of times the application has been run, which we shall call the session number, as well as the session number of the last time an action was used. The value of *UsageCount* is used to look up the maximum number of sessions the item can go unused before it becomes hidden, which is then compared with the difference between the current session number and the session number of the last use of the item. If that difference is greater than the number determined in *PrioritySchedule*, the item is hidden. The default values of *PrioritySchedule* are shown in the table below:

Table 6.2 Default values of the action manager's PrioritySchedule property

Number of sessions in which an action band item was used	Number of sessions an item will remain unhidden after its last use
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29
25 or more	31

It is possible to disable item hiding at design time. To prevent a specific action (and all the collections containing it) from becoming hidden, find its TActionClientItem object and set its *UsageCount* to -1. To prevent hiding for an entire collection of items, such as the File menu or even the main menu bar, find the *TActionClients* object associated with the collection and set its *HideUnused* property to *False*.

Using action lists

Note The contents of this section apply to setting up toolbars and menus for cross-platform development. For Windows development you can also use the methods described here. However, using action bands instead is simpler and offers more options. The action lists will be handled automatically by the Action Manager. See "Organizing actions for toolbars and menus" on page 6-16 for information on using action bands and the Action Manager.

Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for performing actions (for example, when a toolbar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

Setting up action lists

Setting up action lists is fairly easy once you understand the basic steps involved:

- Create the action list.
- Add actions to the action list.
- Set properties on the actions.
- Attach clients to the action.

Here are the steps in more detail:

1. Drop a *TActionList* object onto your form or data module. (ActionList is on the Standard page of the component palette.)
2. Double-click the *TActionList* object to display the Action List editor.
   a. Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.
   b. The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Action Classes dialog box. Select all the standard actions you want to add to the action list and click OK.

      or

   c. Create a new action of your own: right-click and choose New Action.

3 Set the properties of each action in the Object Inspector. (The properties you set affect every client of the action.)

The *Name* property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible,* and *Execute*) correspond to the properties and events of its client controls. The client's corresponding properties are typically, but not necessarily, the same name as the corresponding client property. For example, an action's *Enabled* property corresponds to a *TToolButton*'s *Enabled* property. However, an action's *Checked* property corresponds to a *TToolButton*'s *Down* property.

4 If you use the predefined actions, the action includes a standard response that occurs automatically. If creating your own action, you need to write an event handler that defines how the action responds when fired. See "What happens when an action fires" on page 6-24 for details.

5 Attach the actions in the action list to the clients that require them:

- Click on the control (such as the button or menu item) on the form or data module. In the Object Inspector, the *Action* property lists the available actions.
- Select the one you want.

The standard actions, such as *TEditDelete* or *TDataSetPost*, all perform the action you would expect. You can look at the online reference Help for details on how all of the standard actions work if you need to. If writing your own actions, you'll need to understand more about what happens when the action is fired.

What happens when an action fires

When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.

Responding with events

When a client component or control is clicked or otherwise acted on, a series of events occurs to which you can respond. For example, the following code illustrates the event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
  { Toggle Toolbar1's visibility }
  ToolBar1.Visible := not ToolBar1.Visible;
end;
```

Note For general information about events and event handlers, see Working with events and event handlers"Working with events and event handlers" on page 25.

You can supply an event handler that responds at one of three different levels: the action, the action list, or the application. This is only a concern if you are using a new generic action rather than a predefined standard action. You do not have to worry about this if using the standard actions because standard actions have built-in behavior that executes when these events occur.

The order in which the event handlers will respond to events is as follows:

- Action list
- Application
- Action

When the user clicks on a client control, Delphi calls the action's Execute method which defers first to the action list, then the Application object, then the action itself if neither action list nor Application handles it. To explain this in more detail, Delphi follows this dispatching sequence when looking for a way to respond to the user action:

1 If you supply an *OnExecute* event handler for the action list and it handles the action, the application proceeds.

  The action list's event handler has a parameter called *Handled*, that returns *False* by default. If the handler is assigned and it handles the event, it returns *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
  Handled := True;
end;
```

  If you don't set *Handled* to *True* in the action list event handler, then processing continues.

2 If you did not write an *OnExecute* event handler for the action list or if the event handler doesn't handle the action, the application's *OnActionExecute* event handler fires. If it handles the action, the application proceeds.

  The global *Application* object receives an *OnActionExecute* event if any action list in the application fails to handle an event. Like the action list's *OnExecute* event handler, the *OnActionExecute* handler has a parameter *Handled* that returns *False* by default. If an event handler is assigned and handles the event, it returns *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
  { Prevent execution of all actions in Application }
  Handled := True;
end;
```

3 If the application's *OnExecute* event handler doesn't handle the action, the action's *OnExecute* event handler fires.

You can use built-in actions or create your own action classes that know how to operate on specific target classes (such as edit controls). When no event handler is found at any level, the application next tries to find a target on which to execute the action. When the application locates a target that the action knows how to address, it invokes the action. See the next section for details on how the application locates a target that can respond to a predefined action class.

How actions find their targets

"What happens when an action fires" on page 6-24 describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

The application looks for the target using the following sequence:

1. Active control: The application looks first for an active control as a potential target.
2. Active form: If the application does not find an active control or if the active control can't act as a target, it looks at the screen's *ActiveForm*.
3. Controls on the form: If the active form is not an appropriate target, the application looks at the other controls on the active form for a target.

If no target is located, nothing happens when the event is fired.

Some controls can expand the search to defer the target to an associated component; for example, data-aware controls defer to the associated dataset component. Also, some predefined actions do not use a target; for example, the File Open dialog.

Updating actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
  { Indicate whether ToolBar1 is currently visible }
  (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

Warning Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

Predefined action classes

You can add predefined actions to your application by right-clicking on the Action Manager and choosing New Standard Action. The New Standard Action Classes dialog box is displayed listing the predefined action classes and the associated standard actions. These are actions that are included with Delphi and they are objects that automatically perform actions. The predefined actions are organized within the following classes:

Table 6.3 Action classes

Class	Description
Edit	Standard edit actions: Used with an edit control target. *TEditAction* is the base class for descendants that each override the *ExecuteTarget* method to implement copy, cut, and paste tasks by using the clipboard.
Format	Standard formatting actions: Used with rich text to apply text formatting options such as bold, italic, underline, strikeout, and so on. *TRichEditAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement formatting of the target.
Help	Standard Help actions: Used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command onto a Help system.
Window	Standard window actions: Used with forms as targets in an MDI application. *TWindowAction* is the base class for descendants that each override the *ExecuteTarget* method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
File	File actions: Used with operations on files such as File Open, File Run, or File Exit.
Search	Search actions: Used with search options. *TSearchAction* implements the common behavior for actions that display a modeless dialog where the user can enter a search string for searching an edit control.
Tab	Tab control actions: Used to move between tabs on a tab control such as the Prev and Next buttons on a wizard.
List	List control actions: Used for managing items in a list view.
Dialog	Dialog actions: Used with dialog components. *TDialogAction* implements the common behavior for actions that display a dialog when executed. Each descendant class represents a specific dialog.
Internet	Internet actions: Used for functions such as Internet browsing, downloading, and sending mail.
DataSet	DataSet actions: Used with a dataset component target. *TDataSetAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement navigation and editing of the target. *TDataSetAction* introduces a *DataSource* property that ensures actions are performed on that dataset. If *DataSource* is nil, the currently focused data-aware control is used.
Tools	Tools: Additional tools such as *TCustomizeActionBars* for automatically displaying the customization dialog for action bands.

All of the action objects are described under the action object names in the online reference Help. Refer to the Help for details on how they work.

Writing action components

You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of object. Then, you can use your custom actions in the same way you use pre-defined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler.

Component writers can use the classes in the QStdActns and DBActns units as examples for deriving their own action classes to implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction*, *TWindowAction*, and so on) generally override *HandlesTarget, UpdateTarget,* and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task. These methods are described here:

Method	Description
HandlesTarget	Called automatically when the user invokes an object (such as a toolbutton or menu item) that is linked to the action. The *HandlesTarget* method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the *Target* parameter as a “target”. See “How actions find their targets” on page 6-26 for details.
UpdateTarget	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of *OnUpdateAction*. See “Updating actions” on page 6-26 for details.
ExecuteTarget	Called automatically when the action fires in response to a user action in place of *OnExecute* (for example, when the user selects a menu item or presses a tool button that is linked to this action). See “What happens when an action fires” on page 6-24 for details.

Registering actions

When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the Actnlist unit:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);

procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

When you call *RegisterActions*, the actions you register appear in the Action List editor for use by your applications. You can supply a category name to organize your actions, as well as a *Resource* parameter that lets you supply default property values.

For example, the following code registers the standard actions with the IDE:

```
{ Standard action registration }

RegisterActions('', [TAction], nil);

RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);

RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

When you call *UnRegisterActions*, the actions no longer appear in the Action List editor.

Creating and managing menus

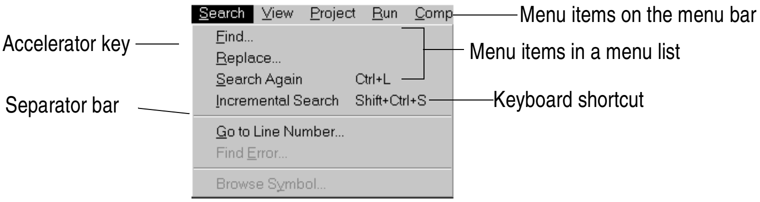
Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This chapter explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer
- Building menus
- Editing menu items in the Object Inspector
- Using the Menu Designer context menu
- Using menu templates
- Saving a menu as a template
- Adding images to menu items

Figure 6.3 Menu terminology

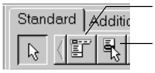


For information about hooking up menu items to the code that executes when they are selected, see “Associating menu events with event handlers” on page 3-28.

Opening the Menu Designer

You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a MainMenu or PopupMenu component to your form. Both menu components are located on the Standard page of the component palette.

Figure 6.4 MainMenu and PopupMenu components



MainMenu component

PopupMenu component

A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then either:

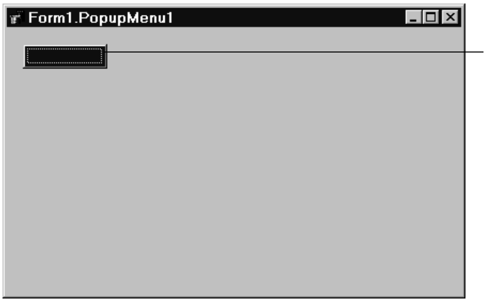
- Double-click the menu component.

  or

- From the Properties page of the Object Inspector, select the *Items* property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

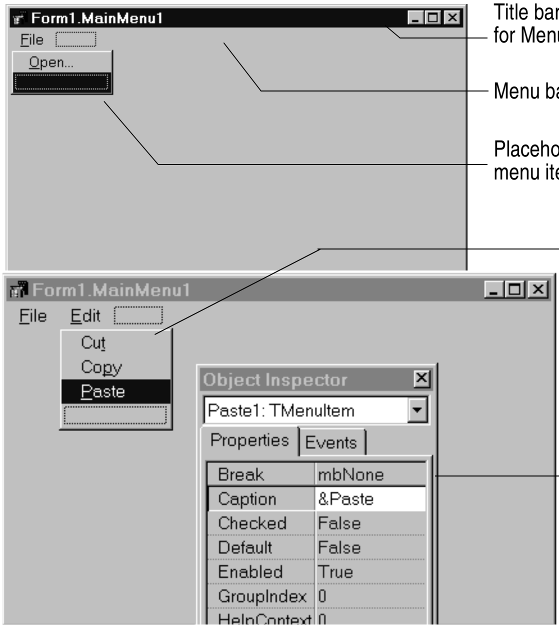
  The Menu Designer appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the Object Inspector.

Figure 6.5 Menu Designer for a pop-up menu



Placeholder for first menu item

Figure 6.6 Menu Designer for a main menu



Break	mbNone
Caption	&Paste
Checked	False
Default	False
Enabled	True
GroupIndex	0
HelpContext	0

Title bar (shows *Name* property for Menu component)

Menu bar

Placeholder for menu item

Menu Designer displays WYSIWYG menu items as you build the menu.

A TMenuItem object is created and the *Name* property set to the menu item *Caption* you specify (minus any illegal characters and plus a numeric suffix).

Building menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

This section discusses the basics of creating a menu at design time. For more information about menu templates, see “Using menu templates” on page 6-38.

Naming menus

As with all components, when you add a menu component to the form, Delphi gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows Object Pascal naming conventions.

Delphi adds the menu name to the form's type declaration, and the menu name then appears in the Component list.

Naming the menu items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type the value for the *Name* property.
- Type the value for the *Caption* property first, and let Delphi derive the *Name* property from the caption.

  For example, if you give a menu item a *Caption* property value of *File*, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi leaves the *Caption* property blank until you type a value.

Note If you enter characters in the *Caption* property that are not valid for Object Pascal identifiers, Delphi modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Delphi precedes the number with a character to derive the *Name* property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

Table 6.4 Sample captions and their derived names

Component caption	Derived name	Explanation
&File	File1	Removes ampersand
&File (2nd occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numerical order
1234 (2nd occurrence)	N12342	Adds a number to disambiguate the derived name
$@@@#	N1	Removes all non-standard characters, adding preceding letter and numerical order
- (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, Delphi adds any menu item names to the form's type declaration, and those names then appear in the Component list.

Adding, inserting, and deleting menu items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time,

1 Select the position where you want to create the menu item.

If you've just opened the Menu Designer, the first position on the menu bar is already selected.

2 Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the Object Inspector and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.

3 Press *Enter*.

The next placeholder for a menu item is selected.

If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Delphi has filled in the *Name* property based on the value you entered for the caption. (See "Naming the menu items" on page 6-32.)

4 Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press *Esc* to return to the menu bar.

Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press *Enter* to complete an action. To return to the menu bar, press *Esc*.

To insert a new, blank menu item,

1 Place the cursor on a menu item.
2 Press *Ins*.

Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command,

1 Place the cursor on the menu item you want to delete.
2 Press *Del*.

Note You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

Adding separator bars

Separator bars insert a line between menu items. You can use separator bars to indicate groupings within the menu list, or simply to provide a visual break in a list.

To make the menu item a separator bar, type a hyphen (-) for the caption.

Specifying accelerator keys and keyboard shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing *Alt+* the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Delphi automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual*.

To specify an accelerator,

- Add an ampersand in front of the appropriate letter.

  For example, to add a Save menu command with the *S* as an accelerator key, type `&Save`.

Keyboard shortcuts enable the user to perform the action without using the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut,

- Use the Object Inspector to enter a value for the *ShortCut* property, or select a key combination from the drop-down list.

  This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

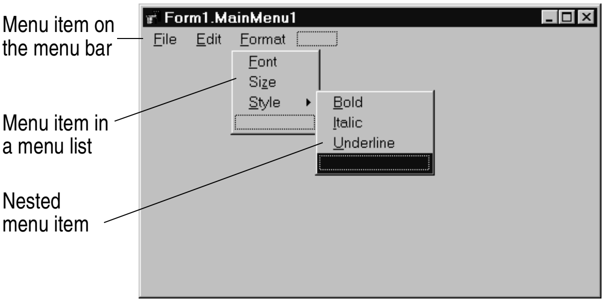
Caution Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

Creating submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Delphi supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

Figure 6.7 Nested menu structures



To create a submenu,

1 Select the menu item under which you want to create a submenu.

2 Press *Ctrl*→ to create the first placeholder, or right-click and choose Create Submenu.

3 Type a name for the submenu item, or drag an existing menu item into this placeholder.

4 Press *Enter*, or ↓, to create the next placeholder.

5 Repeat steps 3 and 4 for each item you want to create in the submenu.

6 Press *Esc* to return to the previous menu level.

Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well—moving a menu item into an existing submenu just creates one more level of nesting.

Moving menu items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a different menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a *different* menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

1 Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.

2 Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list,

1 Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.

2 Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

Adding images to menu items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list.

To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu item.

To add an image to a menu item using an image list:

1 Drop a *TMainMenu* or *TPopupMenu* object on a form.

2 Drop a *TImageList* object on the form.

3 Open the ImageList editor by double clicking on the *TImageList* object.

4 Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.

5 Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.

6 Create your menu items and submenu items as described previously.

7 Select the menu item you want to have an image in the Object Inspector and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

Note Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

Viewing the menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu,

1 If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.

2 If the form has more than one menu, select the menu you want to view from the form's *Menu* property drop-down list.

   The menu appears in the form exactly as it will when you run the program.

Editing menu items in the Object Inspector

This section has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the Object Inspector, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list in the Object Inspector and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items,

1 Switch focus from the Menu Designer window to the Object Inspector by clicking the properties page of the Object Inspector.

2 Close the Menu Designer as you normally would.

  The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

Using the Menu Designer context menu

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to "Using menu templates" on page 6-38.)

To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.

Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

Table 6.5 Menu Designer context menu commands

Menu command	Action
Insert	Inserts a placeholder above or to the left of the cursor.
Delete	Deletes the selected menu item (and all its sub-items, if any).
Create Submenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item.
Select Menu	Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu.
Save As Template	Opens the Save Template dialog box, where you can save a menu for future reuse.
Insert From Template	Opens the Insert Template dialog box, where you can select a template to reuse.
Delete Templates	Opens the Delete Templates dialog box, where you can choose to delete any existing templates.
Insert From Resource	Opens the Insert Menu from Resource file dialog box, where you can choose an .mnu file to open in the current form.

Switching between menus at design time

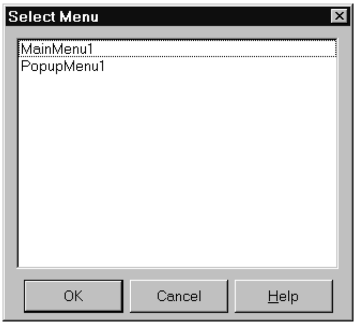
If you're designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch between menus in a form,

1 Right-click in the Menu Designer and choose Select Menu.

  The Select Menu dialog box appears.

Figure 6.8 Select Menu dialog box



This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

2 From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form,

1 Give focus to the form whose menus you want to choose from.
2 From the Component list, select the menu you want to edit.
3 On the Properties page of the Object Inspector, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

Using menu templates

Delphi provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates shipped with Delphi are stored in the BIN subdirectory in a default installation. These files have a .DMT (Delphi menu template) extension.

You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

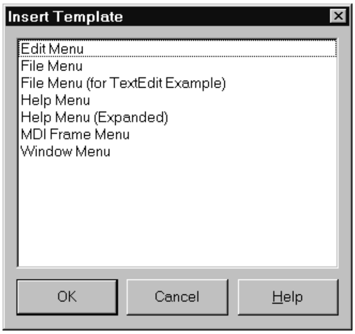
To add a menu template to your application,

1 Right-click the Menu Designer and choose Insert From Template.

(If there are no templates, the Insert From Template option appears dimmed in the context menu.)

The Insert Template dialog box opens, displaying a list of available menu templates.

Figure 6.9 Sample Insert Template dialog box for menus



2 Select the menu template you want to insert, then press *Enter* or choose OK.

This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1 Right-click the Menu Designer and choose Delete Templates.

(If there are no templates, the Delete Templates option appears dimmed in the context menu.)

The Delete Templates dialog box opens, displaying a list of available templates.

2 Select the menu template you want to delete, and press *Del*.

Delphi deletes the template from the templates list and from your hard disk.

Saving a menu as a template

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in your BIN subdirectory as .DMT files.

To save a menu as a template,

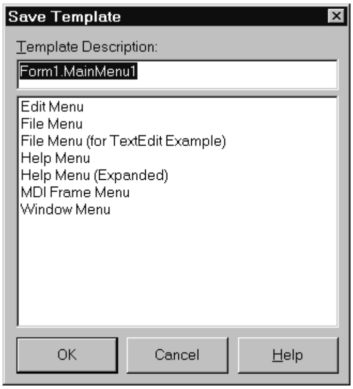
1 Design the menu you want to be able to reuse.

This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.

2 Right-click in the Menu Designer and choose Save As Template.

The Save Template dialog box appears.

Figure 6.10 Save Template dialog box for menus



Save Template

Template Description: Form1.MainMenu1

Edit Menu
File Menu
File Menu (for TextEdit Example)
Help Menu
Help Menu (Expanded)
MDI Frame Menu
Window Menu

3 In the Template Description edit box, type a brief description for this menu, and then choose OK.

The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

Note The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

Naming conventions for template menu items and event handlers

When you save a menu as a template, Delphi does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Delphi names it *File1*. If you insert it into a menu with an existing menu item named *File1*, Delphi names it *File2*.

Delphi also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Delphi still generates the event handler name.

You can easily associate items in the menu template with existing *OnClick* event handlers in the form For more information, see “Associating an event with an existing event handler” on page 3-27.

Manipulating menu items at runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item’s *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the *Enabled* property.

For examples that use the menu item’s *Visible* and *Enabled* properties, see “Disabling menu items” on page 7-10.

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. The following section discusses this in more detail.

Merging menus

For MDI applications, such as the text editor sample application, and for OLE client applications, your application’s main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*. Note that OLE technology is limited to Windows applications only and is not available for use in cross-platform programming.

You prepare menus for merging by specifying values for two properties:

- *Menu*, a property of the form
- *GroupIndex*, a property of menu items in the menu

Specifying the active menu: Menu property

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
Form1.Menu := SecondMenu;
```

Determining the order of merged menu items: GroupIndex property

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:

- Lower numbers appear first (farther left) in the menu.

  For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn’t be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.

- To replace items in the main menu, give items on the child menu the same *GroupIndex* value.

This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well.

Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.

- To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.

  For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

Importing resource files

Delphi supports use of menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your Delphi project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load existing .RC menu files,

1. In the Menu Designer, place your cursor where you want the menu to appear.

   The imported menu can be part of a menu you are designing, or an entire menu in itself.

2. Right-click and choose Insert From Resource.

   The Insert Menu From Resource dialog box appears.

3. In the dialog box, select the resource file you want to load, and choose OK.

   The menu appears in the Menu Designer window.

Note If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

Designing toolbars and cool bars

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added.

Note Cool bars are not available in CLX for cross-platform applications.

You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

You can add a toolbar to a form in several ways:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.

- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.
- Use a cool bar (*TCoolBar*) component and add controls to it. The cool bar displays controls on independently movable and resizable bands.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar and cool bar components, you are ensuring that your application has the look and feel of a Windows application because you are using the native Windows controls. If these operating system controls change in the future, your application could change as well. Also, since the toolbar and cool bar rely on common components in Windows, your application requires the COMCTL32.DLL. Toolbars and cool bars are not supported in WinNT 3.51 applications.

The following sections describe how to

- Add a toolbar and corresponding speed button controls using the panel component
- Add a toolbar and corresponding tool button controls using the Toolbar component
- Add a cool bar using the cool bar component
- Respond to clicks
- Add hidden toolbars and cool bars
- Hide and show toolbars and cool bars

Adding a toolbar using a panel component

To add a toolbar to a form using the panel component,

1. Add a panel component to the form (from the Standard page of the component palette).
2. Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.
3. Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked

- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Add a speed button to a toolbar panel
- Assign a speed button's glyph
- Set the initial condition of a speed button
- Create a group of speed buttons
- Allow toggle buttons

Adding a speed button to a panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional page of the component palette) on the panel.

The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

Assigning a speed button's glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time,

1. Select the speed button.
2. In the Object Inspector, select the *Glyph* property.
3. Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

Setting the initial condition of a speed button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

Table 6.6 lists some actions you can set to change a speed button's appearance:

Table 6.6 Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	*GroupIndex* property to a value other than zero and its *Down* property to *True*.

Table 6.6 Setting speed buttons' appearance (continued)

To make a speed button:	Set the toolbar's:
Appear disabled	*Enabled* property to *False*.
Have a left margin	*Indent* property to a value greater than 0.

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to *True*.

Creating a group of speed buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

Allowing toggle buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a *toggle*. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*.

Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

Adding a toolbar using the toolbar component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not. To add a toolbar to a form using the toolbar component,

1. Add a toolbar component to the form (from the Win32 page of the component palette). The toolbar automatically aligns to the top of the form.
2. Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement tool buttons on a toolbar, do the following:

- Add a tool button
- Assign images to tool buttons
- Set the tool buttons' appearance
- Create a group of tool buttons
- Allow toggled tool buttons

Adding a tool button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the component palette onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

Assigning images to tool buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled. To assign images to tool buttons at design time,

1. Select the toolbar on which the buttons appear.
2. In the Object Inspector, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.
3. Select a tool button.
4. In the Object Inspector, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

Setting tool button appearance and initial conditions

Table 6.7 lists some actions you can set to change a tool button's appearance:

Table 6.7 Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	(on tool button) *Style* property to *tbsCheck* and *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.

Table 6.7 Setting tool buttons’ appearance (continued)

To make a tool button:	Set the toolbar’s:
Have a left margin	*Indent* property to a value greater than 0.
Appear to have “pop-up” borders, thus making the toolbar appear transparent	*Flat* property to *True*.

Note Using the *Flat* property of *TToolBar* requires version 4.70 or later of COMCTL32.DLL.

To force a new row of controls after a specific tool button, Select the tool button that you want to appear last in the row and set its *Wrap* property to *True*.

To turn off the auto-wrap feature of the toolbar, set the toolbar’s *Wrapable* property to *False*.

Creating groups of tool buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is *False*.
- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

Allowing toggled tool buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*.

As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.

Adding a cool bar component

Note The *TCoolBar* component requires version 4.70 or later of COMCTL32.DLL and is not available in CLX.

The cool bar component (*TCoolBar*)—also called a *rebar*—displays windowed controls on independently movable, resizable bands. The user can position the bands by dragging the resizing grips on the left side of each band.

To add a cool bar to a form in a Windows application,

1. Add a cool bar component to the form (from the Win32 page of the component palette). The cool bar automatically aligns to the top of the form.
2. Add windowed controls from the component palette to the bar.

Only VCL components that descend from *TWinControl* are windowed controls. You can add graphic controls—such as labels or speed buttons—to a cool bar, but they will not appear on separate bands.

Setting the appearance of the cool bar

The cool bar component offers several useful configuration options. Table 6.8 lists some actions you can set to change a tool button's appearance:

Table 6.8 Setting a cool button's appearance

To make the cool bar:	Set the toolbar's:
Resize automatically to accommodate the bands it contains	*AutoSize* property to *True.*
Bands maintain a uniform height	*FixedSize* property to *True*.
Reorient to vertical rather than horizontal	*Vertical* property to *True*. This changes the effect of the *FixedSize* property.
Prevent the *Text* properties of the bands from displaying at runtime	*ShowText* property to *False*. Each band in a cool bar has its own *Text* property.
Remove the border around the bar	*BandBorderStyle* to *bsNone.*
Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.)	*FixedOrder* to *True.*
Create a background image for the cool bar	*Bitmap* property to *TBitmap* object.
Choose a list of images to appear on the left of any band	*Images* property to *TImageList* object.

To assign images to individual bands, select the cool bar and double-click on the *Bands* property in the Object Inspector. Then select a band and assign a value to its *ImageIndex* property.

Responding to clicks

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For more information, see “Working with events and event handlers” on page 3-25 and “Generating a handler for a component’s default event” on page 3-26.

Assigning a menu to a tool button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

1 Select the tool button.

2 In the Object Inspector, assign a pop-up menu (*TPopupMenu*) to the tool button’s *DropDownMenu* property.

If the menu’s *AutoPopup* property is set to *True*, it will appear automatically when the button is pressed.

Adding hidden toolbars

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar,

1 Add a toolbar, cool bar, or panel component to the form.
2 Set the component’s *Visible* property to *False*.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

Hiding and showing toolbars

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To hide or show a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
```

Demo programs

For examples of Windows applications that use actions and action lists, refer to Demos\RichEdit. In addition, the Application wizard (File|New Project page), MDI Application, SDI Application, and Winx Logo Applications can use the action and action list objects. For examples of cross-platform applications, refer to Demos\CLX.

Chapter

7

Working with controls

Controls are visual components that the user can interact with at runtime. This chapter describes a variety of features common to many controls.

Implementing drag-and-drop in controls

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another.

- Starting a drag operation
- Accepting dragged items
- Dropping items
- Ending a drag operation
- Customizing drag and drop with a drag object
- Changing the drag mouse pointer

Starting a drag operation

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and, optionally, an integer parameter called *Threshold*. If you pass *True* for *Immediate*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse the number of pixels specified by *Threshold*. Calling

```
BeginDrag False)
```

allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { drag only if left button pressed }
    with Sender as TFileListBox do  { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then  { is there an item here? }
          BeginDrag(False);  { if so, drag it }
    end;
end;
```

Accepting dragged items

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. If *Accept* is *True*, the application sends a drag-and-drop event to the control.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drop. In the following example, a directory tree view accepts dragged items only if they come from a file list box.

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True
  else
    Accept := False;
end;
```

Dropping items

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components as they are passed over.

In the following example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

Ending a drag operation

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the item was dragged. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is **nil**, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In this example, a file list box handles an end-drag event by refreshing its file list.

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;
```

Customizing drag and drop with a drag object

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* or *TDragObjectEx* and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-and-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

TDragObjectEx descendants are freed automatically whereas descendants of *TDragObject* are not. If you have *TDragObject* descendants that you are not explicitly freeing, you can change them so they descend from *TDragObjectEx* instead to prevent memory loss.

Drag objects let you drag items between a form implemented in the application's main executable file and a form implemented using a DLL, or between forms that are implemented using different DLLs.

Changing the drag mouse pointer

You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property (VCL only).

Implementing drag-and-dock in controls

Note Drag and dock properties are available in the VCL but not CLX.

Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel.

- Making a windowed control a docking site
- Making a control a dockable child
- Controlling how child controls are docked
- Controlling how child controls are undocked
- Controlling how child controls respond to drag-and-dock operations

Making a windowed control a docking site

Note Drag and dock properties are available in the VCL but not CLX.

To make a windowed control a docking site,

1. Set the *DockSite* property to *True*.
2. If the dock site object should not appear except when it contains a docked client, set its *AutoSize* property to *True*. When *AutoSize* is *True*, the dock site is sized to 0 until it accepts a child control for docking. Then it resizes to fit around the child control.

Making a control a dockable child

Note Drag and dock properties are available in the VCL but not CLX.

To make a control a dockable child,

1. Set its *DragKind* property to *dkDock*. When *DragKind* is *dkDock*, dragging the control moves the control to a new docking site or undocks the control so that it becomes a floating window. When *DragKind* is *dkDrag* (the default), dragging the control starts a drag-and-drop operation which must be implemented using the *OnDragOver*, *OnEndDrag*, and *OnDragDrop* events.

2 Set its *DragMode* to *dmAutomatic*. When *DragMode* is *dmAutomatic*, dragging (for drag-and-drop or docking, depending on *DragKind*) is initiated automatically when the user starts dragging the control with the mouse. When *DragMode* is *dmManual*, you can still begin a drag-and-dock (or drag-and-drop) operation by calling the *BeginDrag* method.

3 Set its *FloatingDockSiteClass* property to indicate the *TWinControl* descendant that should host the control when it is undocked and left as a floating window. When the control is released and not over a docking site, a windowed control of this class is created dynamically, and becomes the parent of the dockable child. If the dockable child control is a descendant of *TWinControl*, it is not necessary to create a separate floating dock site to host the control, although you may want to specify a form in order to get a border and title bar. To omit a dynamic container window, set *FloatingDockSiteClass* to the same class as the control, and it will become a floating window with no parent.

Controlling how child controls are docked

Note Drag and dock properties are available in the VCL but not CLX.

A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control).

Three events allow docking sites to further constrain how child controls are docked:

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var   InfluenceRect:
TRect; var CanDock: Boolean) of object;
```

OnGetSiteInfo occurs on the docking site when the user drags a dockable child over the control. It allows the site to indicate whether it will accept the control specified by the *DockClient* parameter as a child, and if so, where the child must be to be considered for docking. When *OnGetSiteInfo* occurs, *InfluenceRect* is initialized to the screen coordinates of the docking site, and *CanDock* is initialized to *True*. A more limited docking region can be created by changing *InfluenceRect* and the child can be rejected by setting *CanDock* to *False*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer; State:
TDragState; var Accept: Boolean) of object;
```

OnDockOver occurs on the docking site when the user drags a dockable child over the control. It is analogous to the *OnDragOver* event in a drag-and-drop operation. Use it to signal that the child can be released for docking, by setting the *Accept* parameter. If the dockable control is rejected by the *OnGetSiteInfo* event handler (perhaps because it is the wrong type of control), *OnDockOver* does not occur.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer) of
object;
```

OnDockDrop occurs on the docking site when the user releases the dockable child over the control. It is analogous to the *OnDragDrop* event in a normal drag-and-drop operation. Use this event to perform any necessary accommodations to accepting the control as a child control. Access to the child control can be obtained using the *Control* property of the *TDockObject* specified by the *Source* parameter.

Controlling how child controls are undocked

Note Drag and dock properties are available in the VCL but not CLX.

A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean)   of
object;
```

The *Client* parameter indicates the child control that is trying to undock, and the *Allow* parameter lets the docking site (*Sender*) reject the undocking. When implementing an *OnUnDock* event handler, it can be useful to know what other children (if any) are currently docked. This information is available in the read-only *DockClients* property, which is an indexed array of *TControl*. The number of dock clients is given by the read-only *DockClientCount* property.

Controlling how child controls respond to drag-and-dock operations

Note Drag and dock properties are available in the VCL but not CLX.

Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.

Working with text in controls

The following sections explain how to use various features of rich edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment
- Adding scrollbars at runtime
- Adding the clipboard object
- Selecting text
- Selecting all text
- Cutting, copying, and pasting text
- Deleting selected text
- Disabling menu items
- Providing a pop-up menu
- Handling the OnPopup event

Setting text alignment

In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to.

For example, the following code attaches an OnClick event handler to the Character | Left menu item, then attaches the same event handler to both the Right and Center menu items on the Character menu.

```
procedure TEditForm.AlignClick(Sender: TObject);
begin
  Left1.Checked := False;  { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { check the item clicked }
  with Editor do  { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

Adding scroll bars at runtime

Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word-wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word-wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime,

1. Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.
2. Set the rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a Character | WordWrap menu item.

```
procedure TEditForm.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
    WordWrap := not WordWrap;  { toggle word-wrapping }
    if WordWrap then
      ScrollBars := ssVertical  { wrapped requires only vertical }
    else
```

```
      ScrollBars := ssBoth;  { unwrapped might need both }
      WordWrap1.Checked := WordWrap;  { check menu item to match property }
    end;
  end;
```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

Adding the clipboard object

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. The *Clipboard* object in Delphi encapsulates a clipboard (such as the Windows Clipboard) and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the *Clipbrd* unit.

To add the *Clipboard* object to an application,

1. Select the unit that will use the clipboard.
2. Search for the `implementation` reserved word.
3. Add *Clipbrd* to the `uses` clause below `implementation`.
   - If there is already a `uses` clause in the `implementation` part, add *Clipbrd* to the end of it.
   - If there is not already a `uses` clause, add one that says

```
uses Clipbrd;
```

For example, in an application with a child window, the uses clause in the unit's implementation part might look like this:

```
uses
  MDIFrame, Clipbrd;
```

Selecting text

Before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

Table 7.1 lists properties commonly used to handle selected text.

Table 7.1 Properties of selected text

Property	Description
SelText	Contains a string representing the selected text in the component.
SelLength	Contains the length of a selected string.
SelStart	Contains the starting position of a string.

Selecting all text

The *SelectAll* method selects the entire contents of the rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a rich edit or memo control, call the *RichEdit1* control's *SelectAll* method.

For example,

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
  RichEdit1.SelectAll;  { select all text in RichEdit }
end;
```

Cutting, copying, and pasting text

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard. (See "Using the clipboard with graphics" on page 8-21 for information on using the clipboard with graphics.)

To cut, copy, or paste text with the clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the Edit|Cut, Edit|Copy, and Edit|Paste commands, respectively:

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
```

Deleting selected text

You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

```
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
```

Disabling menu items

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to *False*.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *RichEdit1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);  {enable or disable the Paste
                                                    menu item}
  HasSelection := Editor.SelLength > 0;  { True if text is selected }
  Cut1.Enabled := HasSelection;  { enable menu items if HasSelection is True }
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;
```

The *HasFormat* method of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. By calling *HasFormat* with the parameter *CF_TEXT*, you can determine whether the clipboard contains any text, and enable or disable the Paste item as appropriate.

Chapter 8, "Working with graphics and multimedia" provides more information about using the clipboard with graphics.

Providing a pop-up menu

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's *PopupMenu* property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form,

1 Place a pop-up menu component on the form.

2 Use the Menu Designer to define the items for the pop-up menu.

3 Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.

4 Attach handlers to the *OnClick* events of the pop-up menu items.

Handling the OnPopup event

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu, as described in "Disabling menu items" on page 7-10.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

To adjust menu items on a pop-up menu before displaying them,

1 Select the pop-up menu component.
2 Attach an event handler to its *OnPopup* event.
3 Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *Edit1Click* event handler described previously in "Disabling menu items" on page 7-10 is attached to the pop-up menu component's *OnPopup* event. A line of code is added to *Edit1Click* for each item in the pop-up menu.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
  Paste2.Enabled := Paste1.Enabled;{Add this line}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;{Add this line}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection;{Add this line}
  Delete1.Enabled := HasSelection;
end;
```

Adding graphics to controls

Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and tool bars. Instead of using the standard method of drawing a control or its items, the control's owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see "Adding images to menu items" on page 6-35.

All owner-draw controls contain lists of items. Usually, those lists are lists of strings that are displayed as text, or lists of objects that contain strings that are displayed as text. You can associate an object with each item in a list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in Delphi involves these steps:

1. Indicating that a control is owner-drawn
2. Adding graphical objects to a string list
3. Drawing owner-drawn items

Indicating that a control is owner-drawn

To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and tool bars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as "OnCustomDraw" or "OnAdvancedCustomDraw".

Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing. List views and tab controls have a property called *OwnerDraw* that enables or disabled the default drawing.

List boxes and combo boxes have additional owner-draw styles, called *fixed* and *variable*, as Table 7.2 describes. Other controls are always fixed, although the size of the item that contains the text may vary, the size of each item is determined before drawing the control.

Table 7.2 Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the *ItemHeight* property.	*lbOwnerDrawFixed*, *csOwnerDrawFixed*
Variable	Each item might have a different height, determined by the data at runtime.	*lbOwnerDrawVariable*, *csOwnerDrawVariable*

Adding graphical objects to a string list

Every string list has the ability to hold a list of objects in addition to its list of strings.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

Adding images to an application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example, you can store bitmaps for owner-draw controls in hidden image controls, like this:

1 Add image controls to the main form.
2 Set their *Name* properties.
3 Set the *Visible* property for each image control to *False*.
4 Set the *Picture* property of each image to the desired bitmap using the Picture editor from the Object Inspector.

The image controls are invisible when you run the application.

Adding images to a string list

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate* event handler looks like this:

```
procedure TFMForm.FormCreate(Sender: TObject);
var
  Drive: Char;
  AddedIndex: Integer;
begin
  for Drive := 'A' to 'Z' do  { iterate through all possible drives }
  begin
    case GetDriveType(Drive + ':/') of  { positive values mean valid drives }
     DRIVE_REMOVABLE:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
     DRIVE_FIXED:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
     DRIVE_REMOTE:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
    end;
    if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then  { current drive? }
     DriveTabSet.TabIndex := AddedIndex;  { then make that current tab }
  end;
end;
```

Drawing owner-drawn items

When you indicate that a control is owner-drawn, either by setting a property or supplying a custom draw event handler, the control is no longer drawn on the screen. Instead, the operating system generates events for each visible item in the control. Your application handles the events to draw the items.

To draw the items in an owner-draw control, do the following for each visible item in the control. Use a single event handler for all items.

1 Size the item, if needed.

  Items of the same size (for example, with a list box style of *lsOwnerDrawFixed*), do not require sizing.

2 Draw the item.

Sizing owner-draw items

Before giving your application the chance to draw each item in a variable owner-draw control, the operating system generates a measure-item event. The measure-item event tells the application where the item appears on the control.

Delphi determines the size of the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to be the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be big enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. Depending on the control, the name of the event can vary. List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the size of that item. The size is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer);  { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

Note You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

Drawing owner-draw items

When an application needs to draw or redraw an owner-draw control, the operating system generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a whole or subitems.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing typically start with one of the following:

- *OnDraw*, such as *OnDrawItem* or *OnDrawCell*
- *OnCustomDraw*, such as *OnCustomDrawItem*
- *OnAdvancedCustomDraw*, such as *OnAdvancedCustomDrawItem*

The draw-item event contains parameters identifying the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
  Draw(R.Left, R.Top + 4, Bitmap);  { draw bitmap }
  TextOut(R.Left + 2 + Bitmap.Width,  { position text }
    R.Top + 2, DriveTabSet.Tabs[Index]);  { and draw it to the right of the
                                          bitmap }
  end;
end;
```

Chapter

8

Working with graphics and multimedia

Graphics and multimedia elements can add polish to your applications. Delphi offers a variety of ways to introduce these features into your application. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, Delphi includes special components that can play audio and video clips. Note that multimedia components are not available for cross-platform programming.

Overview of graphics programming

The VCL graphics components defined in the Graphics unit encapsulate the Windows Graphics Device Interface (GDI), making it easy to add graphics to your Windows applications. CLX graphics components defined in the QGraphics unit encapsulate the Qt graphics widgets for adding graphics to cross-platform applications.

To draw graphics in a Delphi application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it takes care of device context, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles (drawings in CLX). Canvases are available only at runtime, so you do all your work with canvases by writing code.

VCL Note Since *TCanvas* is a wrapper resource manager around the Windows device context, you can also use all Windows GDI functions on the canvas. The *Handle* property of the canvas *is* the device context Handle.

CLX Note *TCanvas* is a wrapper resource manager around a Qt painter. The *Handle* property of the canvas *is* typed pointer to an instance of a Qt painter object. Having this instance pointer exposed allows you to use low-level Qt graphics library functions that require an instance pointer to a painter object.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* message (VCL) or event (CLX).

When working with graphics, you often encounter the terms *drawing* and *painting*:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

The examples in the beginning of this chapter demonstrate how to draw various graphics, but they do so in response to *OnPaint* events. Later sections show how to do the same kind of drawing in response to other events.

Refreshing the screen

At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages on Windows, which the VCL routes to *OnPaint* events. (If you are using CLX for cross-platform development, a paint event is generated, which CLX routes to *OnPaint* events.) If you have written an *OnPaint* event handler for that object, it is called when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics or if using the VCL, you want to paint a background on a form.

While some operating systems automatically handle the redrawing of the client area of a window that has been invalidated, Windows does not. In the Windows operating system anything drawn on the screen is permanent. When a form or control is temporarily obscured, for example during window dragging, the form or control must repaint the obscured area when it is re-exposed. For more information about the WM_PAINT message, see the Windows online Help.

If you use the *TImage* control to display a graphical image on a form, the painting and refreshing of the graphic contained in the *TImage* is handled automatically. The *Picture* property specifies the actual bitmap, drawing, or other graphic object that

TImage displays. You can also set the *Proportional* property to ensure that the image can be fully displayed in the image control without any distortion. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*'s canvas maps directly onto the screen device (VCL) or the painter (CLX), so that anything drawn to the *PaintBox*'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for the image to be repainted each time the client area is invalidated.

Types of graphic objects

The VCL/CLX provides the graphic objects shown in Table 8.1. These objects have methods to draw on the canvas, which are described in "Using Canvas methods to draw graphic objects" on page 8-9 and to load and save to graphics files, as described in "Loading and saving graphics files" on page 8-18.

Table 8.1 Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the Picture *Register* method. Use this to handle arbitrary files such as displaying images in an image control.
Bitmap	A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the *handle* is copied, not the image.
Clipboard	Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the clipboard; manage and manipulate formats for objects in the clipboard.
Icon	Represents the value loaded from an icon file (::ICO file).
Metafile (VCL only) Drawing (CLX only)	Contains a file that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Metafiles or drawings are extremely scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, metafiles and drawings do not display as fast as bitmaps. Use a metafile or drawing when versatility or precision is more important than performance.

Common properties and methods of Canvas

Table 8.2 lists the commonly used properties of the Canvas object. For a complete list of properties and methods, see the *TCanvas* component in online Help.

Table 8.2 Common properties of the Canvas object

Properties	Descriptions
Font	Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font.
Brush	Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas.
Pen	Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen.
PenPos	Specifies the current drawing position of the pen.
Pixels	Specifies the color of the area of pixels within the current ClipRect.

These properties are described in more detail in “Using the properties of the Canvas object” on page 8-5.

Table 8.3 is a list of several methods you can use:

Table 8.3 Common methods of the Canvas object

Method	Descriptions
Arc	Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle.
Chord	Draws a closed figure represented by the intersection of a line and an ellipse.
CopyRect	Copies part of an image from another canvas into the canvas.
Draw	Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y).
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
FloodFill (VCL only)	Fills an area of the canvas using the current brush.
FrameRect	Draws a rectangle using the Brush of the canvas to draw the border.
LineTo	Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y).
MoveTo	Changes the current drawing position to the point (X,Y).
Pie	Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas.

Table 8.3 Common methods of the Canvas object (continued)

Method	Descriptions
Polygon	Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point.
Polyline	Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points.
Rectangle	Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use *Rectangle* to draw a box using Pen and fill it using Brush.
RoundRect	Draws a rectangle with rounded corners on the canvas.
StretchDraw	Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.
TextHeight, TextWidth	Returns the height and width, respectively, of a string in the current font. Height includes leading between lines.
TextOut	Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string.
TextRect	Writes a string inside a region; any portions of the string that fall outside the region do not appear.

These methods are described in more detail in “Using Canvas methods to draw graphic objects” on page 8-9.

Using the properties of the Canvas object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This section describes

- Using pens
- Using brushes
- Reading and setting pixels

Using pens

The *Pen* property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change: *Color*, *Width*, *Style*, and *Mode*.

- Color property: Changes the pen color
- Width property: Changes the pen width
- Style property: Changes the pen style
- Mode property: Changes the pen mode

The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

You can use *TPenRecall* for quick saving off and restoring the properties of pens.

Changing the pen color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

Changing the pen width

A pen's width determines the thickness, in pixels, of the lines it draws.

Note When the thickness is greater than 1, Windows 95/98 always draw solid lines, regardless of the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's *Width* property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
 PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
```

Changing the pen style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

VCL Note If developing a cross-platform application for deployment under Windows, Windows 95/98 does not support dashed or dotted line styles for pens wider than one pixel and makes all larger pens solid, no matter what style you specify.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

1 Select all six pen-style buttons and select the Object Inspector | Events | *OnClick* event and in the Handler column, type `SetPenStyle`.

  Delphi generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.

2 Fill in the click-event handler by setting the pen's style depending on the value of *Sender*, which is the control that sent the click event:

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
   else if Sender = DashPen then Style := psDash
   else if Sender = DotPen then Style := psDot
   else if Sender = DashDotPen then Style := psDashDot
   else if Sender = DashDotDotPen then Style := psDashDotDot
   else if Sender = ClearPen then Style := psClear;
  end;
end;
```

Changing the pen mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on. See *TPen* in online Help for details.

Getting the pen position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its *PenPos* property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the *MoveTo* method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

Note Drawing a line with the *LineTo* method also moves the current position to the endpoint of the line.

Using brushes

The *Brush* property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- Color property: Changes the fill color
- Style property: Changes the brush style
- Bitmap property: Uses a bitmap as a brush pattern

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

You can use *TBrushRecall* for quick saving off and restoring the properties of brushes.

Changing the brush color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar (see "Changing the pen color" on page 8-6):

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

Changing the brush style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its *Style* property to one of the predefined values: *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, or *bsDiagCross*.

This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight buttons are selected, the Object Inspector|Events|*OnClick* is set, and the *OnClick* handler is named *SetBrushStyle*. Here is the handler code:

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
```

```
     else if Sender = HorizontalBrush then Style := bsHorizontal
     else if Sender = VerticalBrush then Style := bsVertical
     else if Sender = FDiagonalBrush then Style := bsFDiagonal
     else if Sender = BDiagonalBrush then Style := bsBDiagonal
     else if Sender = CrossBrush then Style := bsCross
     else if Sender = DiagCrossBrush then Style := bsDiagCross;
    end;
  end;
```

Setting the Brush Bitmap property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;
```

Note The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remains valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

Reading and setting pixels

You will notice that every canvas has an indexed *Pixels* property that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color.

Note Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the *TBitmap.ScanLine* property.

Using Canvas methods to draw graphic objects

This section shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines

- Drawing shapes
- Drawing rounded rectangles
- Drawing polygons

Drawing lines and polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

Drawing lines

To draw a straight line on a canvas, use the *LineTo* method of the canvas.

LineTo draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
 begin
   MoveTo(0, 0);
   LineTo(ClientWidth, ClientHeight);
   MoveTo(0, ClientHeight);
   LineTo(ClientWidth, 0);
 end;
end;
```

Drawing polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the *Polyline* method of the canvas.

The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the *MoveTo* method and the *LineTo* method because it eliminates a lot of call overhead.

The following method, for example, draws a rhombus in a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
   Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

This example takes advantage of Delphi's ability to create an open-array parameter on-the-fly. You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter. For more information, see online Help.

Drawing shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This section covers:

- Drawing rectangles and ellipses
- Drawing rounded rectangles
- Drawing polygons

Drawing rectangles and ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's *Rectangle* method or *Ellipse* method, passing the coordinates of a bounding rectangle.

The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
 Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

Drawing rounded rectangles

To draw a rounded rectangle on a canvas, call the canvas's *RoundRect* method.

The first four parameters passed to *RoundRect* are a bounding rectangle, just as for the *Rectangle* method or the *Ellipse* method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

The following method, for example, draws a rounded rectangle in a form's upper left quadrant, rounding the corners as sections of a circle with a diameter of 10 pixels:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

Drawing polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

Polygon takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

For example, the following code draws a right triangle in the lower left half of a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
   Point(ClientWidth, ClientHeight)]);
end;
```

Handling multiple drawing objects in your application

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use
- Changing the tool with speed buttons
- Using drawing tools

Keeping track of which drawing tool to use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time. You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won't be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Object Pascal provides a means to handle both of these shortcomings. You can declare an enumerated type.

An enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Object Pascal's type-checking to ensure that you assign only those specific values.

To declare an enumerated type, use the reserved work type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

For example, the following code declares an enumerated type for each drawing tool available in a graphics application:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

By convention, type identifiers begin with the letter *T*, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for “drawing tool”).

The declaration of the TDrawingTool type is equivalent to declaring a group of constants:

```
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

The main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use Object Pascal's type-checking to prevent many errors. A variable of type TDrawingTool can be assigned only one of the constants dtLine..dtRoundRect. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form's drawing tool:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
 TForm1 = class(TForm)
    ...{ method declarations }
 public
    Drawing: Boolean;
   Origin, MovePt: TPoint;
   DrawingTool: TDrawingTool;{ field to hold current tool }
  end;
```

Changing the tool with speed buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

```
procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;
```

Using drawing tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.

This section describes

- Drawing shapes
- Sharing code among event handlers

Drawing shapes

Drawing shapes is just as easy as drawing lines: Each one takes a single statement; you just need the coordinates.

Here’s a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                            X,Y: Integer);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
       Canvas.LineTo(X, Y)
      end;
   dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
   dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
   dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                   (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
 Drawing := False;
end;
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;
   case DrawingTool of
      dtLine: begin
                Canvas.MoveTo(Origin.X, Origin.Y);
               Canvas.LineTo(MovePt.X, MovePt.Y);
               Canvas.MoveTo(Origin.X, Origin.Y);
               Canvas.LineTo(X, Y);
              end;
     dtRectangle: begin
                      Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                     Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                   end;
     dtEllipse: begin
                    Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                  end;
     dtRoundRect: begin
                     Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                  end;
      end;
     MovePt := Point(X, Y);
    end;
   Canvas.Pen.Mode := pmCopy;
  end;
```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next section shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

Sharing code among event handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form,

1 Add the method declaration to the form object.

You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.

2 Write the method implementation in the implementation part of the form unit.

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

The following code adds a method to the form called *DrawShape* and calls it from each of the handlers. First, the declaration of *DrawShape* is added to the form object's declaration:

```
type
  TForm1 = class(TForm)
    ...{ fields and methods declared here}
 public
    { Public declarations }
   procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

Then, the implementation of *DrawShape* is written in the implementation part of the unit:

```
implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
    Pen.Mode := AMode;
   case DrawingTool of
      dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
          LineTo(BottomRight.X, BottomRight.Y);
         end;
      dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
      dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
      dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
         (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
    end;
  end;
end;
```

The other event handlers are modified to call *DrawShape*.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
 Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
   MovePt := Point(X, Y);{ record the current point }
   DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
  end;
end;
```

Drawing on a graphic

You don’t need any components to manipulate your application’s graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses an image control component to display the graphic on a form.

Once you move the application’s drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes that have been installed such as jpeg graphics.

Note Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control’s canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

Making scrollable graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

Adding an image control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

Note "Adding graphics to controls" on page 7-11 shows how to use graphics in controls.

Placing the control

You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.

Setting the initial bitmap size

When you place an image control, it is simply a container. However, you can set the image control's *Picture* property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in "Loading and saving graphics files" on page 8-18.

To create a blank bitmap when the application starts,

1 Attach a handler to the *OnCreate* event for the form that contains the image.

2 Create a bitmap object, and assign it to the image control's *Picture.Graphic* property.

In this example, the image is in the application's main form, *Form1*, so the code attaches a handler to *Form1*'s *OnCreate* event:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable to hold the bitmap }
begin
  Bitmap := TBitmap.Create;{ construct the bitmap object }
 Bitmap.Width := 200;{ assign the initial width... }
 Bitmap.Height := 200;{ ...and the initial height }
 Image.Picture.Graphic := Bitmap;{ assign the bitmap to the image control }
 Bitmap.Free; {We are done with the bitmap, so free it }
end;
```

Assigning the bitmap to the picture's *Graphic* property copies the bitmap to the picture object. However, the picture object does not take ownership of the bitmap, so after making the assignment, you must free it.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

Drawing on the bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

```
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
  x,y : integer;
  Bitmap : TBitmap;
  P : PByteArray;
begin
  Bitmap := TBitmap.create;
  try
    Bitmap.LoadFromFile('C:\Program Files\Borland\Delphi 4\Images\Splash\256color\
factory.bmp');
    for y := 0 to Bitmap.height -1 do
    begin
      P := Bitmap.ScanLine[y];
      for x := 0 to Bitmap.width -1 do
        P[x] := y;
    end;
  canvas.draw(0,0,Bitmap);
  finally
    Bitmap.free;
  end;
end;
```

Loading and saving graphics files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again.

The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in the following sections:

- Loading a picture from a file
- Saving a picture to a file
- Replacing the picture

Loading a picture from a file

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
  begin
    CurrentFile := OpenPictureDialog1.FileName;
   Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

Saving a picture to a file

The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next section.

The following pair of event handlers, attached to the File|Save and File|Save As menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile){ save if already named }
 else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then{ get a file name }
  begin
```

```
    CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
   Save1Click(Sender);{ then save normally }
  end;
end;
```

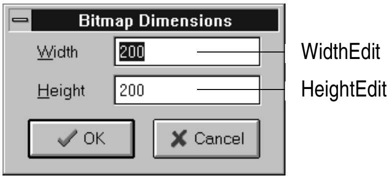
Replacing the picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic (see "Setting the initial bitmap size" on page 8-17), but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box, such as the one in Figure 8.1.

Figure 8.1 Bitmap-dimension dialog box from the BMPDlg unit.



Width 200

Height 200

This particular dialog box is created in the *BMPDlg* unit included with the *GraphEx* project (in the EXAMPLES\DOC\GRAPHEX directory).

With such a dialog box in your project, add it to the uses clause in the unit for your main form. You can then attach an event handler to the File | New menu item's *OnClick* event. Here's an example:

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit;{ make sure focus is on width field }
   WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
   HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
   if ShowModal <> idCancel then{ continue if user doesn't cancel dialog box }
    begin
      Bitmap := TBitmap.Create;{ create fresh bitmap object }
     Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
     Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
     Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
     CurrentFile := '';{ indicate unnamed file }
     Bitmap.Free;
    end;
  end;
end;
```

Note Assigning a new bitmap to the picture object's *Graphic* property causes the picture object to copy the new graphic, but it does not take ownership of it. The picture object maintains its own internal graphic object. Because of this, the previous code frees the bitmap object after making the assignment.

Using the clipboard with graphics

You can use the Windows clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the clipboard object in your application, you must add the Clipbrd (QClipbrd in CLX) unit to the **uses** clause of any unit that needs to access clipboard data.

For cross-platform applications, data that is stored on the clipboard when using CLX is stored as a mime type with an associated *TStream* object. CLX provides the following predefined mime source and mime type string constants for the following CLX objects:

- TBitmap = 'image/delphi.bitmap'
- TComponent = 'application/delphi.component'
- TPicture = 'image/delphi.picture'
- TDrawing = 'image/delphi.drawing'

Copying graphics to the clipboard

You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications.

To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an Edit|Copy menu item:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

Cutting graphics to the clipboard

Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original.

In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the Edit|Cut menu item:

```
procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender);{ copy picture to clipboard }
 with Image.Canvas do
  begin
    CopyMode := cmWhiteness;{ copy everything as white }
   ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
   CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
   CopyMode := cmSrcCopy;{ restore normal mode }
  end;
end;
```

Pasting graphics from the clipboard

If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the clipboard,

1 Call the clipboard's *HasFormat* method (if using the VCL) or *Provides* method (if using CLX) to see whether the clipboard contains a graphic.

HasFormat (or *Provides* in CLX) is a Boolean function. It returns *True* if the clipboard contains an item of the type specified in the parameter. To test for graphics on the Windows platform, you pass *CF_BITMAP*. In cross-platform applications, you pass *SDelphiBitmap*.

2 Assign the clipboard to the destination.

This code shows how to paste a picture from the clipboard into an image control in response to a click on an Edit|Paste menu item:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Windows clipboard? )
 begin
   Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

The same example in CLX for cross-platform development would look as follows:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? )
```

```
    begin
      Image1.Picture.Bitmap.Assign(Clipboard);
    end;
  end;
```

The graphic on the clipboard could come from this application, or it could have been copied from another application, such as Microsoft Paint. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

Rubber banding example

This example describes the details of implementing the "rubber banding" effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The example code in this section is taken from a sample application located in the Demos\DOC\Graphexdirectory. The application draws lines and shapes on a window's canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

The following topics describe the example:

- Responding to the mouse
- Adding a field to a form object to track mouse actions
- Refining line drawing

Responding to the mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This section covers:

- What's in a mouse event
- Responding to a mouse-down action
- Responding to a mouse-up action
- Responding to a mouse move

What's in a mouse event?

The VCL has three mouse events: *OnMouseDown* event, *OnMouseMove* event, and *OnMouseUp* event.

When an application detects a mouse action, it calls whatever event handler you've defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

Table 8.4 Mouse-event parameters

Parameter	Meaning
Sender	The object that detected the mouse action
Button	Indicates which mouse button was involved: *mbLeft*, *mbMiddle*, or *mbRight*
Shift	Indicates the state of the *Alt*, *Ctrl*, and *Shift* keys at the time of the mouse action
X, *Y*	The coordinates where the event occurred

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check *Button* to determine which mouse button caused the event.

Note Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default "primary" and "secondary" mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record *mbLeft* as the value of the *Button* parameter.

Responding to a mouse-down action

Whenever the user presses a button on the mouse, an *OnMouseDown* event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

The VCL generates an empty handler for a mouse-down event on the form:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
end;
```

Responding to a mouse-down action

The following code displays the string 'Here!' at the location on a form clicked with the mouse:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.TextOut(X, Y, 'Here!');{ write text at (X, Y) }
end;
```

When the application runs, you can press the mouse button down with the mouse cursor on the form and have the string, "Here!" appear at the point clicked. This code sets the current drawing position to the coordinates where the user presses the button:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.MoveTo(X, Y);{ set pen position }
end;
```

Pressing the mouse button now sets the pen position, setting the line's starting point. To draw a line to the point where the user releases the button, you need to respond to a mouse-up event.

Responding to a mouse-up action

An *OnMouseUp* event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

Here's a simple *OnMouseUp* event handler that draws a line to the point of the mouse-button release:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
```

This code lets a user draw lines by clicking, dragging, and releasing. In this case, the user cannot see the line until the mouse button is released.

Responding to a mouse move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line to current position }
end;
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

Adding a field to a form object to track mouse actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Delphi adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Delphi "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

The following code gives a form a field called *Drawing* of type Boolean, in the form object's declaration. It also adds two fields to store points *Origin* and *MovePt* of typeTPoint.

```
type
  TForm1 = class(TForm)
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
 public
    Drawing: Boolean;{ field to track whether button was pressed }
    Origin, MovePt: TPoint;{ fields to store points }
  end;
```

When you have a *Drawing* field to track whether to draw, set it to *True* when the user presses the mouse button, and *False* when the user releases it:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;{ set the Drawing flag }
 Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
 Drawing := False;{ clear the Drawing flag }
end;
```

Then you can modify the *OnMouseMove* event handler to draw only when *Drawing* is *True*:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  if Drawing then{ only draw if Drawing flag is set }
   Canvas.LineTo(X, Y);
end;
```

This results in drawing only between the mouse-down and mouse-up events, but you still get a scribbled line that tracks the mouse movements instead of a straight line.

The problem is that each time you move the mouse, the mouse-move event handler calls *LineTo*, which moves the pen position, so by the time you release the button, you've lost the point where the straight line was supposed to start.

Refining line drawing

With fields in place to track various points, you can refine an application's line drawing.

Tracking the origin point

When drawing lines, track the point where the line starts with the *Origin* field.

Origin must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
 Canvas.LineTo(X, Y);
 Drawing := False;
end;
```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions--the application does not yet support "rubber banding."

Tracking movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position,* not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
```

```
      Canvas.LineTo(X, Y);
    end;
  end;
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

MovePt must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);
 MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;{ use XOR mode to draw/erase }
   Canvas.MoveTo(Origin.X, Origin.Y);{ move pen back to origin }
   Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
   Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
   Canvas.LineTo(X, Y);{ draw the new line }
  end;
 MovePt := Point(X, Y);{ record point for next move }
 Canvas.Pen.Mode := pmCopy;
end;
```

Now you get a "rubber band" effect when you draw the line. By changing the pen's mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you're actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final drawing when you release the mouse button.

Working with multimedia

Delphi allows you to add multimedia components to your applications. To do this, you can use either the *TAnimate* component on the Win32 page or the *TMediaPlayer* component on the System page of the Component palette. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application.

For more information on the *TAnimate* and *TMediaPlayer* components, see the VCL on-line help.

The following topics are discussed in this section:

- Adding silent video clips to an application
- Adding audio and/or video clips to an application

Adding silent video clips to an application

The animation control in Delphi allows you to add silent video clips to your application.

To add a silent video clip to an application:

1. Double-click the animate icon on the Win32 page of the Component palette. This automatically puts an animation control on the form window in which you want to display the video clip.
2. Using the Object Inspector, select the *Name* property and enter a new *nam*e for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Delphi identifiers).

   Always work directly with the Object Inspector when setting design time properties and creating event handlers.
3. Do one of the following:
   - Select the *Common AVI* property and choose one of the AVIs available from the drop down list; or
   - Select the *FileName* property and click the ellipsis (…) button, choose an AVI file from any available local or network directories and click Open in the Open AVI dialog; or
   - Select the resource of an AVI using the *ResName* or *ResID* properties. Use *ResHandle* to indicate the module that contains the resource identified by *ResName* or *ResID*.

   This loads the AVI file into memory. If you want to display the first frame of the AVI clip on-screen until it is played using the *Active* property or the *Play* method, then set the *Open* property to *True*.
4. Set the *Repetitions* property to the number of times you want to the AVI clip to play. If this value is 0, then the sequence is repeated until the *Stop* method is called.
5. Make any other changes to the animation control settings. For example, if you want to change the first frame displayed when animation control opens, then set the *StartFrame* property to the desired frame value.
6. Set the *Active* property to *True* using the drop down list or write an event handler to run the AVI clip when a specific event takes place at runtime. For example, to activate the AVI clip when a button object is clicked, write the button's *OnClick* event specifying that. You may also call the *Play* method to specify when to play the AVI.

Note If you make any changes to the form or any of the components on the form after setting *Active* to *True*, the *Active* property becomes *False* and you have to reset it to *True*. Do this either just before runtime or at runtime.

Example of adding silent video clips

Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.

To run this example, create a new project and save the Unit1.pas file as Frmlogo.pas and save the Project1.dpr file as Logo.dpr. Then:

1. Double-click the animate icon from the Win32 page of the Component palette.
2. Using the Object Inspector, set its Name property to *Logo1*.
3. Select its FileName property, click the ellipsis (…) button, choose the cool.avi file from your ..\Demos\Coolstuf directory. Then click Open in the Open AVI dialog.

   This loads the cool.avi file into memory.
4. Position the animation control box on the form by clicking and dragging it to the top right hand side of the form.
5. Set its Repetitions property to 5.
6. Click the form to bring focus to it and set its Name property to *LogoForm1* and its Caption property to *Logo Window*. Now decrease the height of the form to right-center the animation control on it.
7. Double-click the form's *OnActivate* event and write the following code to run the AVI clip when the form is in focus at runtime:

   ```
   Logo1.Active := True;
   ```
8. Double-click the Label icon on the Standard page of the Component palette. Select its Caption property and enter *Welcome to Cool Images 4.0*. Now select its Font property, click the ellipsis (…) button and choose Font Style: Bold, Size: 18, Color: Navy from the Font dialog and click OK. Click and drag the label control to center it on the form.
9. Click the animation control to bring focus back to it. Double-click its *OnStop* event and write the following code to close the form when the AVI file stops:

   ```
   LogoForm1.Close;
   ```
10. Select Run|Run to execute the animated logo window.

Adding audio and/or video clips to an application

The media player component in Delphi allows you to add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software.

Note Audio and video clip support is not provided for cross-platform programming.

To add an audio and/or video clip to an application:

1 Double-click the media player icon on the System page of the Component palette. This automatically put a media player control on the form window in which you want the media feature.

2 Using the Object Inspector, select the *Name* property and enter a new name for your media player control. You will use this when you call the media player control. (Follow the standard rules for naming Delphi identifiers.)

   Always work directly with the Object Inspector when setting design time properties and creating event handlers.

3 Select the *DeviceType* property and choose the appropriate device type to open using the *AutoOpen* property or the *Open* method. (If *DeviceType* is dtAutoSelect the device type is selected based on the file extension of the media file specified by the *FileName* property.) For more information on device types and their functions, see Table 8.5.

4 If the device stores its media in a file, specify the name of the media file using the *FileName* property. Select the *FileName* property, click the ellipsis (…) button, and choose a media file from any available local or network directories and click Open in the Open dialog. Otherwise, insert the hardware the media is stored in (disk, cassette, and so on) for the selected media device, at runtime.

5 Set the *AutoOpen* property to *True*. This way the media player automatically opens the specified device when the form containing the media player control is created at runtime. If *AutoOpen* is *False*, the device must be opened with a call to the *Open* method.

6 Set the *AutoEnable* property to *True* to automatically enable or disable the media player buttons as required at runtime; or, double-click the *EnabledButtons* property to set each button to *True* or *False* depending on which ones you want to enable or disable.

   The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the media player component. The device can also be controlled by the methods that correspond to the buttons (Play, Pause, Stop, Next, Previous, and so on).

7 Position the media player control bar on the form by either clicking and dragging it to the appropriate place on the form or by selecting the *Align* property and choosing the appropriate align position from the drop down list.

   If you want the media player to be invisible at runtime, set the *Visible* property to *False* and control the device by calling the appropriate methods (*Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *Start Recording*, *Eject*).

8 Make any other changes to the media player control settings. For example, if the media requires a display window, set the *Display* property to the control that

displays the media. If the device uses multiple tracks, set the *Tracks* property to the desired track.

Table 8.5 Multimedia device types and their functions

Device Type	Software/Hardware used	Plays	Uses Tracks	Uses a Display Window
dtAVIVideo	AVI Video Player for Windows	AVI Video files	No	Yes
dtCDAudio	CD Audio Player for Windows or a CD Audio Player	CD Audio Disks	Yes	No
dtDAT	Digital Audio Tape Player	Digital Audio Tapes	Yes	No
dtDigitalVideo	Digital Video Player for Windows	AVI, MPG, MOV files	No	Yes
dtMMMovie	MM Movie Player	MM film	No	Yes
dtOverlay	Overlay device	Analog Video	No	Yes
dtScanner	Image Scanner	N/A for Play (scans images on Record)	No	No
dtSequencer	MIDI Sequencer for Windows	MIDI files	Yes	No
dtVCR	Video Cassette Recorder	Video Cassettes	No	Yes
dtWaveAudio	Wave Audio Player for Windows	WAV files	No	No

Example of adding audio and/or video clips (VCL only)

This example runs an AVI video clip of a multimedia advertisement for Delphi. To run this example, create a new project and save the Unit1.pas file to FrmAd.pas and save the Project1.dpr file to DelphiAd.dpr. Then:

1. Double-click the media player icon on the System page of the Component palette.
2. Using the Object Inspector, set the Name property of the media player to *VideoPlayer1*.
3. Select its DeviceType property and choose dtAVIVideo from the drop down list.
4. Select its FileName property, click the ellipsis (…) button, choose the speedis.avi file from your ..\Demos\Coolstuf directory. Click Open in the Open dialog.
5. Set its AutoOpen property to *True* and its Visible property to *False*.
6. Double-click the Animate icon from the Win32 page of the Component palette. Set its AutoSize property to *False*, its Height property to *175* and Width property to *200*. Click and drag the animation control to the top left corner of the form.
7. Click the media player to bring back focus to it. Select its Display property and choose Animate1 from the drop down list.

8 Click the form to bring focus to it and select its Name property and enter *Delphi_Ad*. Now resize the form to the size of the animation control.

9 Double-click the form's *OnActivate* event and write the following code to run the AVI video when the form is in focus:

```
VideoPlayer1.Play;
```

10 Choose Run | Run to execute the AVI video.

Chapter

9

Writing multi-threaded applications

Delphi provides several objects that make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by

- **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.
- **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

Note Not all operating systems implement true multi-processing, even when it is supported by the underlying hardware. For example, Windows 9x only simulates multiprocessing, even if the underlying hardware supports it.

Defining thread objects

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

Note Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the *BeginThread* function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in "Coordinating threads" on page 9-7. For more information on using *BeginThread*, see the online help.

To use a thread object in your application, you must create a new descendant of *TThread*. To create a descendant of *TThread*, choose File | New from the main menu. In the new objects dialog box, select Thread Object. You are prompted to provide a class name for your new thread object. After you provide the name, Delphi creates a new unit file to implement the thread.

Note Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog does not automatically prepend a 'T' to the front of the class name you provide.

The automatically generated unit file contains the skeleton code for your new thread object. If you named your thread *TMyThread*, it would look like the following:

```
unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

You must fill in the code for the *Execute* method. These steps are described in the following sections.

Initializing the thread

If you want to write initialization code for your new thread class, you must override the Create method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the *Priority* property.

If writing a Windows application, *Priority* values fall along a seven-point scale, as described in Table 9.1:

Table 9.1 Thread priorities

Value	Priority
tpIdle	The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with *tpIdle* priority.
tpLowest	The thread's priority is two points below normal.
tpLower	The thread's priority is one point below normal.
tpNormal	The thread has normal priority.
tpHigher	The thread's priority is one point above normal.
tpHighest	The thread's priority is two points above normal.
tpTimeCritical	The thread gets highest priority.

Note If writing a cross-platform application, you must use separate code for assigning priorities on Windows and Linux. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See the CLX version of *TThread* and *Priority* online Help for details.

Warning Boosting the thread priority of a CPU intensive operation may “starve” other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application’s performance:

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
  Priority := tpIdle;
end;
```

Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the *FreeOnTerminate* property to *True*.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to *False* and then explicitly freeing the first thread from the second.

Writing the thread function

The *Execute* method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

Using the main VCL/CLX thread

When you use objects from the VCL or CLX object hierarchies, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main thread is set aside for access of VCL and CLX objects. This is the thread that handles all Windows messages received by components in your application.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's *Synchronize* method. For example:

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;
⋮
procedure TMyThread.Execute;
begin
  ⋮
  Synchronize(PushTheButton);
  ⋮
end;
```

Synchronize waits for the main thread to enter the message loop and then executes the passed method.

Note Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to VCL or CLX objects in console applications.

You do not always need to use the main thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the VCL or CLX thread to enter its message loop. You do not need to use the *Synchronize* method in the following situations:

- Data access components are thread-safe as follows: For BDE-enabled datasets, each thread must have its own database session component. The one exception to this is when you are using Access drivers, which are built using a Microsoft library that is not thread-safe. For dbDirect, as long as the vendor client library is thread-safe, the dbDirect components will be thread-safe. ADO and InterbaseExpress components are thread-safe.

When using data access components, you must still wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset.

For more information about using database sessions with threads in BDE-enabled applications, see "Managing multiple sessions" on page 20-28.

- VisualCLX objects are not thread-safe.
- DataCLX objects are thread-safe.
- Graphics objects are thread-safe. You do not need to use the main VCL or CLX thread to access *TFont*, *TPen*, *TBrush, TBitmap, TMetafile* (VCL only), *TDrawing* (CLX only), or *TIcon*. Canvas objects can be used outside the *Synchronize* method by locking them (see "Locking objects" on page 9-7).
- While list objects are not thread-safe, you can use a thread-safe version, *TThreadList*, instead of *TList*.

Call the *CheckSynchronize* routine periodically within the main thread of your application so that background threads can synchronize their execution with the main thread. The best place to call *CheckSynchronize* is when the application is idle (for example, from an *OnIdle* event handler). This ensures that it is safe to make method calls in the background thread.

Using thread-local variables

Your *Execute* method and any of the routines it calls have their own local variables, just like any other Object Pascal routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section. For example,

```
threadvar
   x : integer;
```

declares an integer type variable that is private to each thread in the application, but global within each thread.

The threadvar section can only be used for global variables. Pointer and Function variables can't be thread variables. Types that use copy-on-write semantics, such as long strings don't work as thread variables either.

Checking for termination by other threads

Your thread begins running when the *Execute* method is called (see "Executing thread objects" on page 9-10) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the *Terminated* property. When another thread tries to terminate your thread, it calls the *Terminate* method. *Terminate* sets your thread's *Terminated* property to *True*. It is up to your *Execute* method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

Handling exceptions in the thread function

The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running.

To catch the exceptions that occur inside your thread function, add a **try**...**except** block to the implementation of the *Execute* method:

```
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
      PerformSomeTask;
  except
    { do something with exceptions }
  end;
end;
```

Writing clean-up code

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an *OnTerminate* event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the *Execute* method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main VCL or CLX thread of your application. This has two implications:

- You can't use any thread-local variables in an *OnTerminate* event handler (unless you want the main VCL or CLX thread values).

- You can safely access any components and VCL or CLX objects from the *OnTerminate* event handler without worrying about clashing with other threads.

For more information about the main VCL or CLX thread, see "Using the main VCL/ CLX thread" on page 9-4.

Coordinating threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

Avoiding simultaneous access

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

Locking objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a *Lock* method that prevents other threads from accessing the canvas until the *Unlock* method is called.

The VCL and CLX also both include a thread-safe list object, *TThreadList*. Calling *TThreadList.LockList* returns the list object while also blocking other execution threads from using the list until the *UnlockList* method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

Using critical sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of *TCriticalSection. TCriticalSection* has two methods, *Acquire* (which blocks other threads from executing the section) and *Release* (which removes the block).

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the *Acquire* method to ensure that no other thread is using it. When finished, threads call the *Release* method so that other threads can access the global memory by calling *Acquire*.

Warning Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling *Acquire* can introduce problems of simultaneous access.

For example, consider an application that has a global critical section variable, *LockXY*, that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

Using the multi-read exclusive-write synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using *TMultiReadExclusiveWriteSynchronizer*. This object acts like a critical section, but allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the *BeginRead* method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the *EndRead* method. Any thread that writes to the protected memory must call *BeginWrite* first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the *EndWrite* method, so that threads waiting to read the memory can begin.

Warning Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

Other techniques for sharing memory

When using objects in the VCL or CLX, use the main thread to execute your code. Using the main thread ensures that the object does not indirectly access any memory that is also used by VCL or CLX objects in other threads. See “Using the main VCL/CLX thread” on page 9-4 for more information on the main thread.

If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See “Using thread-local variables” on page 9-5 for more information about thread-local variables.

Waiting for other threads

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either wait for another thread to completely finish executing, or you can wait for another thread to signal that it has completed a task.

Waiting for a thread to finish executing

To wait for another thread to finish executing, use the *WaitFor* method of that other thread. *WaitFor* doesn't return until the other thread terminates, either by finishing its own *Execute* method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
  end;
  ThreadList1.UnlockList;
end;
```

In the previous example, the list items were only accessed when the *WaitFor* method indicated that the list was successfully filled. This return value must be assigned by the *Execute* method of the thread that was waited for. However, because threads that call *WaitFor* want to know the result of thread execution, not code that calls *Execute*, the *Execute* method does not return any value. Instead, the *Execute* method sets the *ReturnValue* property. *ReturnValue* is then returned by the *WaitFor* method when it is called by other threads. Return values are integers. Your application determines their meaning.

Waiting for a task to be completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (*TEvent*) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. *SetEvent* turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the *ResetEvent* method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don't know which thread will finish last, you can't simply use the *WaitFor* method of one of the threads. Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the *OnTerminate* event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
  ⋮
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter);   { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
  CounterGuard.Release; { release the lock on the counter }
  ⋮
end;
```

The main thread initializes the Counter variable, launches the task threads, and waits for the signal that they are all done by calling the *WaitFor* method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from Table 9.2.

Table 9.2 WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the timeout period elapsed.
wrError	An error occurred while waiting.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ now continue with the main thread. All task threads have finished }
```

Note If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

Executing thread objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the *Resume* method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to *False***.** For example, the following line creates a thread and starts its execution:

```
SecondProcess := TMyThread.Create(false); {create and run the thread }
```

Warning Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

Overriding the default priority

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in "Initializing the thread" on page 9-2. However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

```
SecondProcess :=  TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.Resume; { now run the thread }
```

Note If writing a cross-platform application, you must use separate code for assigning priorities on Windows and Linux. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See the CLX version of *TThread* and *Priority* online Help for details.

Starting and stopping threads

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its *Suspend* method. When it is safe for the thread to resume, call its *Resume* method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*.

You can request that a thread end execution prematurely by calling the *Terminate* method. *Terminate* sets the thread's *Terminated* property to *True*. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is *True*.

Debugging multi-threaded applications

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose View | Threads from the main menu.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

For additional details on the Thread Status box, see online Help.

Chapter

10

Using CLX for cross-platform development

You can use Delphi to develop cross-platform 32-bit applications that run on both the Windows and Linux operating systems. To do this, you can start with an existing Windows application and modify it, or you can create a new application by following the recommended practices for writing platform-independent code. Kylix is Borland's Delphi for Linux software that allows you to compile and develop applications on Linux. If you want to develop and deploy applications on Linux and Windows, you'll need to use Kylix as well as Delphi.

This chapter describes how to change Delphi applications so they will compile on Linux and includes information on the differences between developing applications on Windows and Linux. It also provides guidelines for writing code that is portable between the different environments.

Note Most applications developed using CLX (with no operating system specific API calls) will run on both Linux and Windows platforms. The application must be compiled on the platform on which you want it to run.

Creating cross-platform applications

You create cross-platform applications much as you create any Delphi application. You need to use CLX visual components, and you should not use operating system specific APIs if you want the application to be completely cross-platform. (See "Writing portable code" on page 10-17 for tips on writing cross-platform applications.)

To create a cross-platform application:

1 In the IDE, choose File | New | CLX application.
The Component palette shows components that can be used in CLX applications.

Note Some Windows only nonvisual components can be used in CLX applications. The Component palette includes ADO, BDE, System, DataSnap, InterBase, Site Express, FastNet, QReport, COM+, BizSnap, and Servers tabs which include functionality that will only work in Windows CLX applications. If you plan to compile your application on Linux as well, do not use the components on these tabs or use **$IFDEF**s to mark these sections of the code as Windows only.

2. Develop your application within the IDE. Remember to use only CLX components in your application.
3. Compile and test the application on each platform on which you want to run the application. Review any error messages to see where additional changes need to be made.

When moving an application to Kylix, you need to reset your project options. That's because the .dof file which stores the project options is recreated on Kylix and called .kof (with the default options set). You can also store many of the compiler options with the application by typing Ctrl+O+O. The options are placed at the beginning of the currently open file.

The form file in cross-platform applications will have an extension of xfm instead of dfm. This is to distinguish cross-platform forms that use CLX components from forms that use VCL components. An xfm form file will work on both Windows or Linux but a dfm form only works on Windows.

You could also begin development of a cross-platform application by starting on Kylix instead of Delphi:

1. Develop, compile and test the application on Linux using Kylix.
2. Move the application source files over to Windows.
3. Reset your project options.
4. Recompile the application on Windows using Delphi.

For information on writing platform-independent database or internet applications, see "Cross-platform database applications" on page 10-23 and "Cross-platform Internet applications" on page 10-29.

Porting VCL applications to CLX

If you have Delphi applications that were written for the Windows environment, you can make them cross platform. How easy it will be depends on the nature and complexity of the application and how many Windows dependencies there are.

The following sections describe some of the major differences between the Windows and Linux environments and provide guidelines on how to get started porting an application.

Porting techniques

The following are different approaches you can take to port an application from one platform to another:

Table 10.1 Porting techniques

Technique	Description
Platform-specific port	Targets an operating system and underlying APIs
Cross-platform port	Targets a cross-platform API
Windows emulation	Leave the code alone and port the API it uses

Platform-specific ports

Platform-specific ports tend to be time-consuming, expensive, and only produce a single targeted result. They create different code bases, which makes them particularly difficult to maintain. However, each port is designed for the specific operating system and can take advantage of platform-specific functionality. So, the application typically runs faster.

Cross-platform ports

Cross-platform ports generally provide the quickest technique and the ported applications target multiple platforms. In reality, the amount of work involved in developing cross-platform applications is highly dependent on the existing code. If code has been developed without regard for platform independence, you may run into scenarios where platform-independent “logic” and platform-dependent “implementation” are mixed together.

The cross-platform approach is the preferable approach because business logic is expressed in platform-independent terms. Some services are abstracted behind an internal interface that looks the same on all platforms, but has a specific implementation on each. Delphi’s runtime library is an example of this: The interface is very similar on both platforms, although the implementation may be vastly different. You should separate cross-platform parts, then implement specific services on top. In the end, this approach is the least expensive solution, because of reduced maintenance costs due to a largely shared source base and an improved application architecture.

Windows emulation ports

Windows emulation is the most complex method and it can be very costly, but the resulting Linux application will look most similar to an existing Windows application. This approach involves implementing Windows functionality on Linux. From an engineering point of view, this is solution is very hard to maintain.

Where you want to emulate Windows APIs, you can include two distinct sections using **$IFDEF**s to indicate sections of the code that apply specifically to Windows or Linux.

Porting your application

If you are porting an application that you want to run on both Windows and Linux, you need to modify your code or use **$IFDEF**s to indicate sections of the code that apply specifically to Windows or Linux.

Follow these general steps to port your VCL application to CLX:

1. Open the project containing the application you want to change in Delphi.
2. Copy .dfm files to .xfm files of the same name (for example, rename unit1.dfm to unit1.xfm). Rename (or **$IFDEF**) the reference to the .dfm file in the unit file(s) from `{$R *.dfm}` to `{$R *.xfm}`. (The .xfm file will work in both Kylix and Delphi.)

   For example, change the form reference in the **implementation** section from

   ```
   {$R *.dfm}
   ```

   to

   ```
   {$R *.xfm}
   ```

3. Change (or **$IFDEF**) all **uses** clauses so they refer to the correct units in CLX. (See "CLX and VCL unit comparison" on page 10-9 for information.)

   For example, change the following **uses** clause in a Windows application

   ```
   uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs,
   StdCtrls;
   ```

   to the following for a CLX application:

   ```
   uses Windows, Messages, SysUtils, Variants, Classes, QForms, QControls, QStdCtrls;
   ```

4. Save the project and reopen it. Now the Component palette shows components that can be used in CLX applications.

Note Some Windows only nonvisual components can be used in CLX applications. The Component palette includes ADO, BDE, System, DataSnap, InterBase, Internet Express, Site Express, FastNet, QReport, COM+, Web Services, and Servers tabs which include functionality that will only work in Windows CLX applications. If you plan to compile your application on Linux as well, do not use the components on these tabs or use **$IFDEF**s to mark these sections of the code as Windows only.

5. Rewrite any code that does not require Windows dependencies making the code more platform-independent. Do this using the runtime library routines and constants. (See "Writing portable code" on page 10-17 for information.)
6. Find equivalent functionality for features that are different on Linux. Use **$IFDEF**s (sparingly) to delimit Windows-specific information. (See "Using conditional directives" on page 10-18 for information.)

   For example, you can **$IFDEF** platform-specific code in your source files:

   ```
   [$IFDEF MSWINDOWS]
   IniFile.LoadfromFile('c:\x.txt');
   [$ENDIF]

   [$IFDEF LINUX]
   IniFile.LoadfromFile('/home/name/x.txt');
   [$ENDIF]
   ```

7 Search for references to pathnames in all the project files.

- Pathnames in Linux use a forward slash / as a delimiter (for example, /usr/lib) and files may be located in different directories on the Linux system. Use the PathDelim constant (in SysUtils) to specify the path delimiter that is appropriate for the system. Determine the correct location for any files on Linux.
- Change references to drive letters (for example, C:\) and code that looks for drive letters by looking for a colon at position 2 in the string. Use the DriveDelim constant (in SysUtils) to specify the location in terms that are appropriate for the system.
- In places where you specify multiple paths, change the path separator from semicolon (;) to colon (:). Use the PathSep constant (in SysUtils) to specify the path separator that is appropriate for the system.
- Because file names are case-sensitive in Linux, make sure that your application doesn’t change the case of file names or assume a certain case.

8 Compile, text and debug your application.

To transfer the application to Linux:

1 Move your Delphi Windows application source files and other project-related files onto your Linux computer. (You can share source files between Linux and Windows if you want the program to run on both platforms. Or you can transfer the files using a tool such as ftp using the ASCII mode.)

   Source files should include your unit files (.pas files), project file (.dpr file), and any package files (.dpk files). Project-related files include form files (.xfm files), resource files (.res files), and project options files (.dof files–in Kylix these change to .kof files). If you want to compile your application from the command line only (rather than using the IDE), you’ll need the configuration file (.cfg file–in Kylix this changes to .conf).

2 Open the project in Kylix. You will receive warnings on Windows-specific features that are in use.

3 Compile the project using Kylix. Review any error messages to see where additional changes need to be made.

CLX versus VCL

Kylix uses the Borland Component Library for Cross Platform (CLX) in place of the Visual Component Library (VCL). Within the VCL, many controls provide an easy way to access Windows controls. Similarly, CLX provides access to Qt widgets (from window + gadget) in the Qt shared libraries. Delphi includes both CLX and the VCL.

CLX looks much like the VCL. Most of the component names are the same, many properties have the same names. In addition, CLX, as well as the VCL, will be available on Windows (check the latest release of Delphi to determine availability).

CLX components can be grouped into the following parts:

Table 10.2 CLX parts

Part	Description
VisualCLX	Native cross-platform GUI components and graphics. The components in this area may differ on Linux and Windows.
DataCLX	Client data-access components. The components in this area are a subset of the local, client/server, and n-tier based on client datasets. The code is the same on Linux and Windows.
NetCLX	Internet components including Apache DSO and CGI Web Broker. These are the same on Linux and Windows.
RTL	Runtime Library up to and including Classes.pas. The code is the same on Linux and Windows.

Widgets in VisualCLX replace Windows controls. In CLX, *TWidgetControl* replaces the VCL's *TWinControl*. Other components (such as *TScrollingWidget*) have corresponding names. However, you do not need to change occurrences of *TWinControl* to *TWidgetControl*. Type declarations, such as the following

```
TWinControl = TWidgetControl;
```

appear in the QControls.pas source file to simplify sharing of source code. *TWidgetControl* and its descendants all have a *Handle* property that is a reference to the Qt object; and a *Hooks* property, which is a reference to the hook objects that handle the event mechanism.

Unit names and locations of some classes are different for CLX. You will need to modify **uses** clauses to eliminate references to units that don't exist in CLX and to change the names to CLX units. (Most project files and the interface sections of most units contain a **uses** clause. The implementation section of a unit can also contain its own **uses** clause.)

What CLX does differently

Although much of CLX is implemented so that it is consistent with the VCL, some features are implemented differently. This section provides an overview of some of the differences between CLX and VCL implementations to be aware of when writing cross-platform applications.

Look and feel

The visual environment in Linux looks somewhat different than it does in Windows. The look of dialogs may differ depending on which window manager is in use (for example, if using KDE or Gnome).

Styles

Application-wide "styles" can be used in addition to the *OwnerDraw* properties. You can use the *TApplication.Style* property to specify the look and feel of an application's graphical elements. Using styles, a widget or an application can take on a whole new look. You can still use owner draw on Linux but using styles is recommended.

Variants

All of the variant/safe array code that was in System is in two new units:

- Variants.pas
- VarUtils.pas

The operating system dependent code is now isolated in VarUtils.pas, and it also contains generic versions of everything needed by Variants.pas. If you are converting a VCL application that included Windows calls to a CLX application, you need to replace these calls to calls into VarUtils.pas.

If you want to use variants, you must include the Variants unit to your **uses** clause.

VarIsEmpty does a simple test against *varEmpty* to see if a variant is clear, and on Linux you need to use the *VarIsClear* function to clear a variant.

Custom variant data handler

You can define custom data types for variants. This introduces operator overloading while the type is assigned to the variant. To create a new variant type, descend from the class, *TCustomVariantType*, and instantiate your new variant type.

For an example, see VarCmplx.pas. This unit implements complex mathematics support via custom variants. It supports the following variant operations: addition, subtraction, multiplication, division (not integer division), and negation. It also handles conversion to and from: SmallInt, Integer, Single, Double, Currency, Date, Boolean, Byte, OleStr, and String. Any of the float/ordinal conversion will lose any imaginary portion of the complex value.

Registry

Linux does not use a registry to store configuration information. Instead, you use text configuration files and environment variables instead of using the registry. System configuration files on Linux are often located in /etc, for example, /etc/hosts. Other user profiles are located in hidden files (preceded with a dot), such as .bashrc, which holds bash shell settings or .XDefaults, which is used to set defaults for X programs.

Registry-dependent code may be changed to using a local configuration text file instead stored, for example, in the same directory as the application. Writing a unit containing all the registry functions but diverting all output to a local configuration file is one way you could handle a former dependency on the registry.

To place information in a global location on Linux, you could store a global configuration file in the root directory. This makes it so all of your applications can access the same configuration file. However, you must be sure that the file permissions and access rights are set up correctly.

You can also use ini files in cross-platform applications. However, in CLX, you need to use *TMemIniFile* instead of *TRegIniFile.*

Other differences

CLX implementation also has some other differences that affect the way your application works. This section describes some of those differences.

ToggleButton doesn't get toggled by the Enter key. Pressing Enter doesn't simulate a click event on Kylix as it does in Delphi.

TColorDialog does not have a *TColorDialog.Options* property to set. Therefore, you cannot customize the appearance and functionality of the color selection dialog. Also, *TColorDialog* is not always modal. You can manipulate the title bar of an application with a modal dialog on Kylix (that is, you can select the parent form of the color dialog and do things like maximizing it while the color dialog is open).

At runtime, combo boxes work differently on Kylix than they do in Delphi. On Kylix (but not on Delphi), you can add a item to a drop down by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

TCustomEdit does not implement *Undo*, *ClearUndo*, or *CanUndo.* So there is no way to programmatically undo edits. But application users can undo their edits in an edit box (*TEdit*) at runtime by right-clicking on the edit box and choosing the Undo command.

The key value in a *OnKeyDown* event or *KeyUp* event for the Enter key on Windows is 13. On Linux, this value is 4100. If you check for a hardcoded numeric value for a key, such as checking for a value of 13 for the Enter key, you need to change this when porting a Delphi application to Kylix.

Additional differences exist. Refer to the CLX online documentation for details on all of the CLX objects or in versions of Delphi that include the source code you can refer to the code, it is located in ..\Delphi6\Source\VCL\CLX.

Missing in CLX

When using CLX instead of the VCL, many of the objects are the same. However, the objects may be missing some features (such as properties, methods, or events). The following general features are missing in CLX:

- Bi-directional properties (*BidiMode*) for right-to-left text output or input
- Generic bevel properties on common controls (note that some objects still have bevel properties)
- Docking properties and methods
- Backward compatibility features such components on the Win3.1 tab and *Ctl3D*
- *DragCursor* and *DragKind* (but drag and drop is included)

Features that will not port

Some Windows-specific features supported on Delphi will not transport directly to Linux environments. Features, such as COM, ActiveX, OLE, BDE, and ADO are dependent on Windows technology and are not available in Kylix. The following table lists features that are different on the two platforms and lists the equivalent Kylix feature, if one is available.

Table 10.3 Changed or different features

Delphi/Windows feature	Kylix/Linux feature
ADO components	Regular database components
Automation Servers	Not available
BDE	dbExpress and regular database components
COM+ components (including ActiveX)	Not available
DataSnap	Not yet available
FastNet	Not available
Internet Express	Not yet available
Legacy components (such as items on the Win 3.1 Component palette tab)	Not available
Messaging Application Programming Interface (MAPI) includes a standard library of Windows messaging functions.	SMTP/POP3 let you send, receive, and save email messages
Quick Reports	Not available
Web Services (SOAP)	Not yet available
WebSnap	Not yet available
Windows API calls	CLX methods, Qt calls, libc calls, or calls to other system libraries
Windows messaging	Qt events
Winsock	BSD sockets

The Linux equivalent of Windows DLLs are shared object libraries (.so files), which contain position-independent code (PIC). This has the following consequences:

- Variables referring to an absolute address in memory (using the **absolute** directive) are not allowed.
- Global memory references and calls to external functions are made relative to the EBX register, which must be preserved across calls.

You only need to worry about global memory references and calls to external functions if using assembler—Kylix or Delphi generates the correct code. (For information, see “Including inline assembler code” on page 10-20.)

Kylix library modules and packages are implemented using .so files.

CLX and VCL unit comparison

All of the objects in the VCL or CLX are defined in unit files (.pas source files). For example, you can find the implementation of *TObject* in the System unit, and the Classes unit defines the base *TComponent* class. When you drop an object onto a form or use an object within your application, the name of the unit is added to the **uses** clause which tells the compiler which units to link into the project.

This section provides tables that list the CLX units and the comparable VCL unit, list the units that are for CLX only, and list the units that are for VCL only.

The following table lists VCL units and the comparable CLX units:

Table 10.4 VCL and equivalent CLX units

VCL units	CLX units
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts, and RTLConsts
Contnrs	Contnrs
Controls	QControls
DateUtils	DateUtils
DB	DB
DBActns	QDBActns
DBClient	DBClient
DBCommon	DBCommon
DBConnAdmin	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs
DSIntf	DSIntf
ExtCtrls	QExtCtrls
FMTBCD	FMTBCD
Forms	QForms
Graphics	QGraphics
Grids	QGrids
HelpIntfs	HelpIntfs
ImgList	QImgList
IniFiles	IniFiles
Mask	QMask
MaskUtils	MaskUtils

Table 10.4 VCL and equivalent CLX units (continued)

VCL units	CLX units
Masks	Masks
Math	Math
Menus	QMenus
Midas	Midas
MidConst	MidConst
Printers	QPrinters
Provider	Provider
Qt	Qt
Search	QSearch
Sockets	Sockets
StdActns	QStdActns
StdCtrls	QStdCtrls
SqlConst	SqlConst
SqlExpr	SqlExpr
SqlTimSt	SqlTimSt
SyncObjs	SyncObjs
SysConst	SysConst
SysInit	SysInit
System	System
SysUtils	SysUtils
Types	Types and QTypes
TypInfo	TypInfo
Variants	Variants
VarUtils	VarUtils

The following units are in CLX but not VCL:

Table 10.5 Units in CLX, not VCL

Unit	Description
DirSel	Directory selection
QStyle	GUI look and feel

The following Windows VCL units are not included in CLX mostly because they concern Windows-specific features that are not available on Linux such as ADO, COM, and the BDE. The reason for the unit's exclusion is listed.

Table 10.6 VCL-only units

Unit	Reason for exclusion
ADOConst	No ADO feature
ADODB	No ADO feature
AppEvnts	No TApplicationEvent object

Table 10.6 VCL-only units (continued)

Unit	Reason for exclusion
AxCtrls	No COM feature
BdeConst	No BDE feature
ComStrs	No COM feature
ConvUtils	New feature for Delphi 6
CorbaCon	No Corba feature
CorbaStd	No Corba feature
CorbaVCL	No Corba feature
CtlPanel	No Windows Control Panel support
DataBkr	May appear later in upsell
DBCGrids	No BDE feature
DBExcept	No BDE feature
DBInpReq	No BDE feature
DBLookup	Obsolete
DbOleCtl	No COM feature
DBPWDlg	No BDE feature
DBTables	No BDE feature
DdeMan	No DDE feature
DRTable	No BDE feature
ExtActns	New feature to Delphi 6
ExtDlgs	No picture dialogs
FileCtrl	Obsolete
ListActns	New feature to Delphi 6
MConnect	No COM feature
Messages	Windows-specific area
MidasCon	Obsolete
MPlayer	Windows-specific media player
Mtsobj	No COM feature
MtsRdm	No COM feature
Mtx	No COM feature
mxConsts	No COM feature
ObjBrkr	May appear later in upsell
OleConstMay	No COM feature
OleCtnrs	No COM feature
OleCtrls	No COM feature
OLEDB	No COM feature
OleServer	No COM feature
Outline	Obsolete
Registry	Windows-specific registry support
ScktCnst	Replaced by Sockets
ScktComp	Replaced by Sockets

Table 10.6 VCL-only units (continued)

Unit	Reason for exclusion
SConnect	Unsupported connection protocols
StdConvs	New feature to Delphi 6
SvcMgr	NT Services support
Tabnotbk	Obsolete
Tabs	Obsolete
ToolWin	No docking feature
VarCmplx	New feature to Delphi 6
VarConv	New feature to Delphi 6
VCLCom	No COM feature
WebConst	Windows-specific constants
Windows	Windows-specific (API)

Differences in CLX object constructors

When a CLX object is created, either implicitly in the Forms Designer by placing that object on the form or explicitly in code by using the *Create* method of the object, an instance of the underlying associated widget is created also. The instance of the widget is owned by this CLX object. When the CLX object is deleted by calling the *Free* method or automatically deleted by the CLX object's parent container, the underlying widget is also deleted. This is the same type of functionality that you see in the VCL in Windows applications.

When you explicitly create a CLX object in code, by calling into the Qt interface library such as QWidget_Create(), you are creating an instance of a Qt widget that is not owned by a CLX object. This passes the instance of an existing Qt widget to the CLX object to use during its construction. This CLX object does not own the Qt widget that is passed to it. Therefore, when you call the *Free* method after creating the object in this manner, only the CLX object is destroyed and not the underlying Qt widget instance. This is different from the VCL.

Some CLX objects let you assume ownership of the underlying widget using the *OwnHandle* method. After calling *OwnHandle*, if you delete the CLX object, the underlying widget is destroyed as well.

Sharing source files between Windows and Linux

If you want your application to run on both Windows and Linux, you can share the source files making them accessible to both operating systems. You can do this many ways such as placing the source files on a server that is accessible to both computers or by using Samba on the Linux machine to provide access to files through Microsoft network file sharing for both Linux and Windows. You can choose to keep the source on Linux and create a shared drive on Linux. Or you can keep the source on Windows and create a share on Windows for the Linux machine to access.

You can continue to develop and compile the file on Kylix using objects that are supported by both VCL and CLX. When you are finished, you can compile on both Linux and Windows.

Form files (.dfm files in Delphi) are called .xfm files in Kylix. If you create a new CLX application in Delphi or Kylix, an .xfm is created instead of a .dfm. If you plan to write cross-platform applications, the .xfm will work both on Delphi and Kylix.

Environmental differences between Windows and Linux

Currently, cross-platform means an application that can run virtually unchanged on both the Windows and Linux operating systems. The following table lists some of the differences between Linux and the Windows operating environments.

Table 10.7 Differences in the Linux and Windows operating environments

Difference	Description
File name case sensitivity	In Linux, a capital letter is *not* the same as a lowercase letter. The file Test.txt is *not* the same file as test.txt. You need to pay close attention to capitalization of file names on Linux.
Line ending characters	On Windows, lines of text are terminated by CR/LF (that is, ASCII 13 + ASCII 10), but on Linux it is LF. While the code editor in Kylix can handle the difference, you should be aware of this when importing code from Windows.
End of file character	In DOS and Windows, the character value #26 (Ctrl-Z) is treated as the end of the text file, even if there is data in the file after that character. Linux has no special end of file character; the text data ends at the end of the file.
Batch files/shell scripts	The Linux equivalent of .bat files are shell scripts. A script is a text file containing instructions, saved and made executable with the command, `chmod +x <scriptfile>`. To execute it, type its name. (The scripting language depends on the shell you are using on Linux. Bash is commonly used.)
Command confirmation	In DOS or Windows, if you try to delete a file or folder, it asks for confirmation ("Are you sure you want to do that?"). Generally, Linux won't ask; it will just do it. This makes it easy to accidentally destroy a file or the entire file system. There is no way to undo a deletion on Linux unless a file is backed up on another media.
Command feedback	If a command succeeds on Linux, it redisplays the command prompt without a status message.
Command switches	Linux uses a dash (-) to indicate command switches or a double dash (--) for multiple character options where DOS uses a slash (/) or dash (-).

Table 10.7 Differences in the Linux and Windows operating environments (continued)

Difference	Description
Configuration files	On Windows, configuration is done in the registry or in files such as autoexec.bat. On Linux, configuration files are created as hidden files starting with a dot (.). Many are placed in the /etc directory and your home directory. Linux also uses environment variables such as LD_LIBRARY_PATH (search path for libraries). Other important environment variables: HOME Your home directory (/home/sam) TERM Terminal type (xterm, vt100, console) SHELL Path to your shell (/bin/bash) USER Your login name (sfuller) PATH List to search for programs They are specified in the shell or in rc files such as the .bashrc.
DLLs	On Linux, you use shared object files (.so). In Windows, these are dynamic link libraries (DLLs).
Drive letters	Linux doesn't have drive letters. An example Linux pathname is /lib/security. See DriveDelim in the runtime library.
Exceptions	Operating system exceptions are called signals on Linux.
Executable files	On Linux, executable files require no extension. On Windows, executable files have an exe extension.
File name extensions	Linux does not use file name extensions to identify file types or to associate files with applications.
File permissions	On Linux, files (and directories) are assigned read, write, and execute permissions for the file owner, group, and others. For example, `-rwxr-xr-x` means, from left to right: `-` is the file type (`-` = ordinary file, `d` = directory, `l` = link); `rwx` are the permissions for the file owner (read, write, execute); `r-x` are the permissions for the group of the file owner (read, execute); and `r-x` are the permissions for all other users (read, execute). The root user (superuser) can override these permissions. You need to make sure that your application runs under the correct user and has proper access to required files.
Make utility	Borland's make utility is not available on the Linux platform. Instead, you can use Linux's own GNU make utility.
Multitasking	Linux fully supports multitasking. You can run several programs (in Linux, called processes) at the same time. You can launch processes in the background (using & after the command) and continue working straight away. Linux also lets you have several sessions.
Pathnames	Linux uses a forward slash (/) wherever DOS uses a backslash (\). A PathDelim constant can be used to specify the appropriate character for the platform. See PathDelim in the runtime library.
Search path	When executing programs, Windows always checks the current directory first, then looks at the PATH environment variable. Linux never looks in the current directory but searches only the directories listed in PATH. To run a program in the current directory, you usually have to type ./ before it. You can also modify your PATH to include ./ as the first path to search.

Table 10.7 Differences in the Linux and Windows operating environments (continued)

Difference	Description
Search path separator	Windows uses the semicolon as a search path separator. Linux uses a colon. See PathDelim in the runtime library.
Symbolic links	On Linux, a symbolic link is a special file that points to another file on disk. Place symbolic links in the global bin directory that points to your application's main files and you don't have to modify the system search path. A symbolic link is created with the ln (link) command. Windows has shortcuts for the GUI desktop. To make a program available at the command line, Windows install programs typically modify the system search path.

Directory structure on Linux

Directories are different in Linux. Any file or device can be mounted anywhere on the file system.

Note Linux pathnames use forward slashes as opposed to Windows use of backslashes. The initial slash stands for the root directory.

Following are some commonly used directories in Linux.

Table 10.8 Common Linux directories

Directory	Contents
/	The root or top directory of the entire Linux file system
/root	The root file system; the Superuser's home directory
/bin	Commands, utilities
/sbin	System utilities
/dev	Devices shown as files
/lib	Libraries
/home/username	Files owned by the user where username is the user's login name.
/opt	Optional
/boot	Kernel that gets called when the system starts up
/etc	Configuration files
/usr	Applications, programs. Usually includes directories like /usr/spool, /usr/man, /usr/include, /usr/local
/mnt	Other media mounted on the system such as a CD or a floppy disk drive
/var	Logs, messages, spool files
/proc	Virtual file system and reporting system statistics
/tmp	Temporary files

Note Different distributions of Linux sometimes place files in different locations. A utility program may be placed in /bin in a Red Hat distribution but in /usr/local/bin in a Debian distribution.

Refer to www.pathname.com for additional details on the organization of the UNIX/Linux hierarchical file system and to read the *Filesystem Hierarchy Standard*.

Writing portable code

If you are writing cross-platform applications that are meant to run on Windows and Linux, you can write code that compiles under different conditions. Using conditional compilation, you can maintain your Windows coding, yet also make allowances for Linux operating system differences.

To create applications that are easily portable between Windows and Linux, remember to

- reduce or isolate calls to platform-specific (Win32 or Linux) APIs; use CLX methods instead.
- eliminate Windows messaging (PostMessage, SendMessage) constructs within an application.
- use *TMemIniFile* instead of *TRegIniFile*.
- observe and preserve case-sensitivity in file and directory names.
- port any external assembler TASM code. The GNU assembler, “as,” does not support the TASM syntax. (See “Including inline assembler code” on page 10-20.)

Try to write the code to use platform-independent runtime library routines and use constants found in System, SysUtils, and other runtime library units. For example, use the PathDelim constant to insulate your code from ‘/’ versus ‘\’ platform differences.

Another example involves the use of multibyte characters on both platforms. Windows code traditionally expects only 2 bytes per multibyte character. In Linux, multibyte character encoding can have many more bytes per char (up to 6 bytes for UTF-8). Both platforms can be accommodated using the StrNextChar function in SysUtils. Existing Windows code such as the following

```
while p^ <> #0 do
begin
   if p^ in LeadBytes then
      inc(p);
   inc(p);
end;
```

can be replaced with platform-independent code like this:

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

This example is platform portable and supports multibyte characters longer than 2 bytes, but still avoids the performance cost of a procedure call for non-multibyte locales.

If using runtime library functions is not a workable solution, try to isolate the platform-specific code in your routine into one chunk or into a subroutine. Try to limit the number of **$IFDEF** blocks to maintain source code readability and portability. The conditional symbol WIN32 is not defined on Linux. The conditional symbol LINUX is defined, indicating the source code is being compiled for the Linux platform.

Using conditional directives

Using **$IFDEF** compiler directives is a reasonable way to conditionalize your code for the Windows and Linux platforms. However, because **$IFDEF**s make source code harder to understand and maintain, you need to understand when it is reasonable to use **$IFDEF**s. When considering the use of **$IFDEF**s, the top questions should be "Why does this code require an **$IFDEF**?" and "Can this be written without an **$IFDEF**?"

Follow these guidelines for using **$IFDEF**s within cross-platform applications:

- Try not to use **$IFDEF**s unless absolutely necessary. **$IFDEF**s in a source file are only evaluated when source code is compiled. Unlike C/C++, Delphi does not require unit sources (header files) to compile a project. Full rebuilds of all source code is an uncommon event for most Delphi projects.
- Do not use **$IFDEF**s in package (.dpk) files. Limit their use to source files. Component writers need to create two design-time packages when doing cross-platform development, not one package using **$IFDEF**s.
- In general, use **$IFDEF** MSWINDOWS to test for any Windows platform including WIN32. Reserve the use of **$IFDEF** WIN32 for distinguishing between specific Windows platforms, such as 32-bit versus 64-bit Windows. Don't limit your code to WIN32 unless you know for sure that it will not work in WIN64.
- Avoid negative tests like **$IFNDEF** unless absolutely required. **$IFNDEF** LINUX is *not* equivalent to **$IFDEF** MSWINDOWS.
- Avoid **$IFNDEF/$ELSE** combinations. Use a positive test instead (**$IFDEF**) for better readability.
- Avoid **$ELSE** clauses on platform-sensitive **$IFDEF**s. Use separate **$IFDEF** blocks for LINUX- and MSWINDOWS-specific code instead of **$IFDEF** LINUX/**$ELSE** or **$IFDEF** MSWINDOWS/**$ELSE**.

  For example, old code may contain

  ```
  {$IFDEF WIN32}
     (32-bit Windows code)
  {$ELSE}
     (16-bit Windows code)   //!! By mistake, Linux could fall into this code.
  {$ENDIF}
  ```

  For any non-portable code in **$IFDEF**s, it is better for the source code to fail to compile than to have the platform fall into an **$ELSE** clause and fail mysteriously at runtime. Compile failures are easier to find than runtime failures.

- Use the **$IF** syntax for complicated tests. Replace nested **$IFDEF**s with a boolean expression in an **$IF** directive. You should terminate the **$IF** directive using **$IFEND**, not **$ENDIF**. This allows you to place **$IF** expressions within **$IFDEF**s to hide the new **$IF** syntax from previous compilers.

All of the conditional directives are documented in the online Help. Also see, the topic "Conditional Compilation" in Help for more information.

Terminating conditional directives

Use the **$IFEND** directive to terminate **$IF** and **$ELSEIF** conditional directives. This allows **$IF/$IFEND** blocks to be hidden from older compilers inside of using **$IFDEF/ $ENDIF**. Older compilers won't recognize the **$IFEND** directive. **$IF** can only be terminated with **$IFEND**. You can only terminate old-style directives (**$IFDEF, $IFNDEF, $IFOPT**) with **$ENDIF**.

Note When nesting an **$IF** inside of **$IFDEF/$ENDIF**, do not use **$ELSE** with the **$IF**. Older compilers will see the **$ELSE** and think it is part of the **$IFDEF**, producing a compile error down the line. You can use {**$ELSEIF** True} as a substitute for {**$ELSE**} in this situation, since the **$ELSEIF** won't be taken if the **$IF** is taken first, and the older compilers won't know **$ELSEIF**. Hiding **$IF** for backwards compatibility is primarily an issue for third party vendors and application developers who want their code to run on several different versions.

$ELSEIF is a combination of **$ELSE** and **$IF**. The **$ELSEIF** directive allows you to write multi-part conditional blocks where only one of the conditional blocks will be taken. For example:

```
{$IFDEF doit}
   do_doit
{$ELSEIF  RTLVersion >= 14}
    goforit
{$ELSEIF  somestring = 'yes'}
    beep
{$ELSE}
    last chance
{$IFEND}
```

Of these four cases, only one is taken. If none of the first three conditions is true, the **$ELSE** clause is taken. **$ELSEIF** must be terminated by **$IFEND**. **$ELSEIF** cannot appear after **$ELSE**. Conditions are evaluated top to bottom like a normal **$IF...$ELSE** sequence. In the example, if doit is not defined, RTLVersion is 15, and somestring = 'yes', only the "goforit" block will be taken not the "beep" block, even though the conditions for both are true.

If you forget to use an **$ENDIF** to end one of your **$IFDEF**s, the compiler reports the following error message at the end of the source file:

```
Missing ENDIF
```

If you have more than a few **$IF/$IFDEF** directives in your source file, it can be difficult to determine which one is causing the problem. Kylix or Delphi reports the following error message on the source line of the last **$IF/$IFDEF** compiler directive with no matching **$ENDIF/$IFEND**:

```
Unterminated conditional directive
```

You can start looking for the problem at that location.

Emitting messages

The **$MESSAGE** compiler directive allows source code to emit hints, warnings, and errors just as the compiler does.

```
{$MESSAGE  HINT|WARN|ERROR|FATAL 'text string' }
```

The message type is optional. If no message type is indicated, the default is HINT. The text string is required and must be enclosed in single quotes.

Examples:

`{$MESSAGE 'Boo!'}` emits a hint.

`{$Message Hint 'Feed the cats'}` emits a hint.

`{$Message Warn 'Looks like rain.'}` emits a warning.

`{$Message Error 'Not implemented'}` emits an error, continues compiling.

`{$Message Fatal 'Bang. Yer dead.'}` emits an error, terminates the compiler.

Including inline assembler code

If you include inline assembler code in your Windows applications, you may not be able to use the same code on Linux because of position-independent code (PIC) requirements on Linux. Linux shared object libraries (DLL equivalents) require that all code be relocatable in memory without modification. This primarily affects inline assembler routines that use global variables or other absolute addresses, or that call external functions.

For units that contain only Object Pascal code, the compiler automatically generates PIC when required. PIC units have a .dpu extension (instead of .dcu). It's a good idea to compile every Pascal unit source file into both PIC and non-PIC formats; use the -p compiler switch to generate PIC. Precompiled units are available in both forms.

You may want to code assembler routines differently depending on whether you'll be compiling to an executable or a shared library; use {**$IFDEF** PIC} to branch the two versions of your assembler code. Or you can consider rewriting the routine in Object Pascal to avoid the issue.

Following are the PIC rules for inline assembler code:

- PIC requires all memory references be made relative to the EBX register, which contains the current module's base address pointer (in Linux called the Global Offset Table or GOT). So, instead of

  ```
  MOV EAX,GlobalVar
  ```

  use

  ```
  MOV EAX,[EBX].GlobalVar
  ```

- PIC requires that you preserve the EBX register across calls into your assembly code (same as on Win32), and also that you restore the EBX register *before* making calls to external functions (different from Win32).
- While PIC code will work in base executables, it may slow the performance and generate more code. You don't have any choice in shared objects, but in executables you probably still want to get the highest level of performance that you can.

Messages and system events

Message loops and events work differently on Linux and in CLX, but this primarily affects component writing. Most component and property editors port easily. *TObject.Dispatch* and message method syntax on classes work fine on Linux; under Linux, however, operating system notifications are handled using system events rather than messages.

To create an event handler in a cross-platform application, you can override one of the methods described in Table 10.9 to write your own custom message instead of responding to Windows messages. In the override, call the inherited method so any default processes still take place.

Table 10.9 TWidgetControl protected methods for responding to system events

Method	Description
ChangeBounds	Used when a *TWidgetControl* is resized. Roughly analogous to WM_SIZE or WM_MOVE in Windows. Qt sets the “geometry” of a widget based on the client area, VCL uses the entire control size, which includes what Qt refers to as the frame.
ChangeScale	Called automatically when resizing controls. Used to change the scale of a form and all its controls for a different screen resolution or font size. Because ChangeScale modifies the control’s Top, Left, Width, and Height properties, it changes the position of the control and its children as well as their size.
ColorChanged	Called when the color of the control has been changed.
CursorChanged	Called when the cursor changes shape. The mouse cursor assumes this shape when it's over this widget.
EnabledChanged	Called when an application changes the enabled state of a window or control.
FontChanged	Called when the collection of font resources changed. It sets the font for the widget and informs all children about the change. Roughly analogous to the WM_FONTCHANGE message.
PaletteChanged	Called when the system palette has been changed. .
ShowHintChanged	Called when Help hints are displayed or hidden on a control.
StyleChanged	Called when the window or control’s GUI styles have changed.
TabStopChanged	Called when the tab order on the form has been changed.
VisibleChanged	Called when a control is hidden or shown.
WidgetDestroyed	Called when a widget underlying a control is destroyed.

Qt is a C++ toolkit, so all of its widgets are C++ objects. CLX is written in Object Pascal, and Object Pascal does not interact directly with C++ objects. In addition, Qt uses multiple inheritance in a few places. So Delphi includes an interface layer that converts all of the Qt classes to a series of straight C functions. These are then wrapped in a shared object in Linux and a DLL in Windows.

Every *TWidgetControl* has *CreateWidget*, *InitWidget*, and *HookEvents* virtual methods that almost always have to be overridden. *CreateWidget* creates the Qt widget, and assigns the Handle to the FHandle private field variable. *InitWidget* gets called after the widget is constructed, and the Handle is valid.

Some property assignments in Delphi CLX have moved from the Create constructor to *InitWidget*. This will allow delayed construction of the Qt object until it's really needed. For example, say you have a property named *Color*. In SetColor, you can check with HandleAllocated to see if you have a Qt handle. If the Handle is allocated, you can make the proper call to Qt to set the color. If not, you can store the value in a private field variable, and, in *InitWidget*, you set the property.

Linux supports two types of events: Widget and System. *HookEvents* is a virtual method that hooks the CLX controls event methods to a special hook object that communicates with the Qt object. The hook object is really just a set of method pointers. System events on Kylix go through *EventHandler*, which is basically a replacement for *WndProc*.

Programming differences on Linux

The Linux wchar_t widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal widechars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal widechar data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

In Linux, widestrings are reference counted like long strings (in Windows, they're not).

Multibyte handling differs in Linux. In Windows, multibyte characters (MBCS) are represented as 1- and 2-byte char codes. In Linux, they are represented in 1 to 6 bytes.

AnsiStrings can carry multibyte character sequences, dependent upon the user's locale settings. The Linux encoding for multibyte characters such as Japanese, Chinese, Hebrew, and Arabic may not be compatible with the Windows encoding for the same locale. Unicode is portable, whereas multibyte is not.

In Linux, you cannot use variables on absolute addresses. The syntax `var X: Integer absolute $1234;` is not supported in PIC and is not allowed in Delphi.

Cross-platform database applications

On Windows, Delphi provides several choices for how to access database information. These include using ADO, the Borland Database Engine (BDE), and InterBase Express. These three choices are not available on Kylix, however. Instead, you can use *dbExpress*, a new, cross-platform data access technology, which is also available on Windows, starting with Delphi version 6.

Before you port a database application to *dbExpress* so that it will run on Linux, you should understand the differences between using *dbExpress* and the data access mechanism you were using. These differences occur at different levels.

- At the lowest level, there is a layer that communicates between your application and the database server. This could be ADO, the BDE, or the InterBase client software. This layer is replaced by *dbExpress*, which is a set of lightweight drivers for dynamic SQL processing.
- The low-level data access is wrapped in a set of components that you add to data modules or forms. These components include database connection components, which represent the connection to a database server, and datasets, which represent the data fetched from the server. Although there are some very important differences, due to the unidirectional nature of *dbExpress* cursors, the differences are less pronounced at this level, because datasets all share a common ancestor, as do database connection components.
- At the user-interface level, there are the fewest differences. CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. The major differences at the user interface level arise from changes needed to accommodate the use of cached updates.

For information on porting existing database applications to *dbExpress*, see “Porting database applications to Linux” on page 10-25. For information on designing new *dbExpress* applications, see Chapter 14, “Designing database applications.”

dbExpress differences

On Linux, *dbExpress* manages the communication with database servers. *dbExpress* consists of a set of lightweight drivers that implement a set of common interfaces. Each driver is a shared object (.so file) that must be linked to your application. Because *dbExpress* is designed to be cross-platform, it will also be available on Windows as a set of dynamic-link libraries (.dlls).

As with any data-access layer, *dbExpress* requires the client-side software provided by the database vendor. In addition, it uses a database-specific driver, plus two configuration files, dbxconnections and dbxdrivers. This is markedly less than you need for, say, the BDE, which requires the main Borland Database Engine library (Idapi32.dll) plus a database-specific driver and a number of other supporting libraries.

Here are some other differences between *dbExpress* and the other data-access layers from which you need to port your application:

- *dbExpress* allows for a simpler and faster path to remote databases. As a result, you can expect a noticeable performance increase for simple, straight-through data access.
- *dbExpress* can process queries and stored procedures, but does not support the concept of opening tables.
- *dbExpress* returns only unidirectional cursors.
- *dbExpress* has no built-in update support other than the ability to execute an INSERT, DELETE, or UPDATE query.
- *dbExpress* does no metadata caching, and the design time metadata access interface is implemented using the core data-access interface.
- *dbExpress* executes only queries requested by the user, thereby optimizing database access by not introducing any extra queries.
- *dbExpress* manages a record buffer or a block of record buffers internally. This differs from the BDE, where clients are required to allocate the memory used to buffer records.
- *dbExpress* does not support local tables that are not SQL-based (such as Paradox, dBase, or FoxPro).
- *dbExpress* drivers exist for InterBase, Oracle, DB2, and MySQL. If you are using a different database server, you must either port your data to one of these databases, write a *dbExpress* driver for the database server you are using, or obtain a third-party *dbExpress* driver for your database server.

Component-level differences

When you write a *dbExpress* application, it requires a different set of data-access components than those used in your existing database applications. The *dbExpress* components share the same base classes as other data-access components (*TDataSet* and *TCustomConnection*), which means that many of the properties, methods, and events are the same as the components used in your existing applications.

Table 10.10 lists some of the important database components used in InterBase Express, BDE, and ADO in the Windows environment and shows the comparable *dbExpress* components for use on Linux and in cross-platform applications.

Table 10.10 Comparable data-access components

InterBase Express components	BDE components	ADO components	dbExpress components
TIBDatabase	*TDatabase*	*TADOConnection*	*TSQLConnection*
TIBTable	*TTable*	*TADOTable*	*TSQLTable*
TIBQuery	*TQuery*	*TADOQuery*	*TSQLQuery*
TIBStoredProc	*TStoredProc*	*TADOStoredProc*	*TSQLStoredProc*
TIBDataSet		*TADODataSet*	*TSQLDataSet*

The *dbExpress* datasets (*TSQLTable*, *TSQLQuery*, *TSQLStoredProc*, and *TSQLDataSet*) are more limited than their counterparts, however, because they do not support editing and only allow forward navigation. For details on the differences between the *dbExpress* datasets and the other datasets that are available on Windows, see Chapter 22, "Using unidirectional datasets.".

Because of the lack of support for editing and navigation, most *dbExpress* applications do not work directly with the *dbExpress* datasets. Rather, they connect the *dbExpress* dataset to a client dataset, which buffers records in memory and provides support for editing and navigation. For more information about this architecture, see "Database architecture" on page 14-5.

Note For very simple applications, you can use *TSQLClientDataSet* instead of a *dbExpress* dataset connected to a client dataset. This has the benefit of simplicity, because there is a 1:1 correspondence between the dataset in the application you are porting and the dataset in the ported application, but is less flexible that explicitly connecting a *dbExpress* dataset to a client dataset. For most applications, it is recommended that you use a *dbExpress* dataset connected to a *TClientDataSet* component.

User interface-level differences

CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. As a result, porting the user-interface portion of your database applications introduces few additional considerations beyond those involved in porting any Windows application to CLX.

The major differences at the user interface level arise from differences in the way *dbExpress* datasets or client datasets supply data.

If you are using only *dbExpress* datasets, then you must adjust your user interface to accommodate the fact that the datasets do not support editing and only support forward navigation. Thus, for example, you may need to remove controls that allow users to move to a previous record. Because *dbExpress* datasets do not buffer data, you can't display data in a data-aware grid: only one record can be displayed at a time.

If you have connected the *dbExpress* dataset to a client dataset, then the user interface elements associated with editing and navigation should still work. You need only reconnect them to the client dataset. The main consideration in this case is handling how updates are written to the database. By default, most datasets on Windows write updates to the database server automatically when they are posted (for example, when the user moves to a new record). Client datasets, on the other hand, always cache updates in memory. For information on how to accommodate this difference, see "Updating data in dbExpress applications" on page 10-27.

Porting database applications to Linux

Porting your database application to *dbExpress* allows you to create a cross-platform application that runs both on Windows and Linux. The porting process involves making changes to your application because the technology is different. How difficult it is to port depends on the type of application it is, how complex it is, and what it needs to accomplish. An application that heavily uses Windows-specific technologies such as ADO will be more difficult to port than one that uses Delphi database technology.

Follow these general steps to port your Windows/VCL database application to Kylix/CLX:

1. Consider where database data is stored. *dbExpress* provides drivers for Oracle, Interbase, DB2, and MySQL. The data needs to reside on one of these SQL servers.

   Some versions of Delphi include the Data Pump utility which you can use to move local database data from platforms such as Paradox, dBase, and FoxPro onto one of the supported platforms. (See the datapump.hlp file in Program Files\Common Files\Borland\Shared\BDE for information on using the utility.)

2. If you have not isolated your user interface forms from data modules containing the datasets and connection components, you may want to consider doing so before you start the port. That way, you isolate the portions of your application that require a completely new set of components into data modules. Forms that represent the user interface can then be ported like any other application. For details, see "Porting your application" on page 10-4.

   The remaining steps assume that your datasets and connection components are isolated in their own data modules.

3. Create a new data module to hold the CLX versions of your datasets and connection components.

4. For each dataset in the original application, add a *dbExpress* dataset, *TDataSetProvider* component, and *TClientDataSet* component. Use the correspondences in Table 10.10 to decide which *dbExpress* dataset to use. Give these components meaningful names.

   - Set the *ProviderName* property of the *TClientDataSet* component to the name of the *TDataSetProvider* component.
   - Set the *DataSet* property of the *TDataSetProvider* component to the *dbExpress* dataset.
   - Change the *DataSet* property of any data source components that referred to the original dataset so that it now refers to the client dataset.

5. Set properties on the new dataset to match the original dataset:

   - If the original dataset was a *TTable*, *TADOTable*, or *TIBTable* component, set the new *TSQLTable*'s *TableName* property to the original dataset's *TableName*. Also copy any properties used to set up master/detail relationships or specify indexes. Properties specifying ranges and filters should be set on the client dataset rather than the new *TSQLTable* component.
   - If the original dataset was a *TQuery*, *TADOQuery*, or *TIBQuery* component, set the new *TSQLQuery* component's *SQL* property to the original dataset's *SQL* property. Set the *Params* property of the new *TSQLQuery* to match the value of the original dataset's *Params* or *Parameters* property. If you have set the *DataSource* property to establish a master/detail relationship, copy this as well.

- If the original dataset was a *TStoredProc*, *TADOStoredProc*, or *TIBStoredProc* component, set the new *TSQLStoredProc* component's *StoredProcName* to the *StoredProcName* or *ProcedureName* property of the original dataset. Set the *Params* property of the new *TSQLStoredProc* to match the value of the original dataset's *Params* or *Parameters* property.

6. For any database connection components in the original application (*TDatabase*, *TIBDatabase*, or *TADOConnection*), add a *TSQLConnection* component to the new data module. You must also add a *TSQLConnection* component for every database server to which you connected without a connection component (for example, by using the *ConnectionString* property on an ADO dataset or by setting the *DatabaseName* property of a BDE dataset to a BDE alias).
7. For each *dbExpress* dataset placed in step 4, set its *SQLConnection* property to the *TSQLConnection* component that corresponds to the appropriate database connection.
8. On each *TSQLConnection* component, specify the information needed to establish a database connection. To do so, double-click the *TSQLConnection* component to display the Connection Editor and set parameter values to indicate the appropriate settings. If you had to transfer data to a new database server in step 1, then specify settings appropriate to the new server. If you are using the same server as before, you can look up some of this information on the original connection component:

   - If the original application used *TDatabase*, you must transfer the information that appears in the *Params* and *TransIsolation* properties.
   - If the original application used *TADOConnection*, you must transfer the information that appears in the *ConnectionString* and *IsolationLevel* properties.
   - If the original application used *TIBDatabase*, you must transfer the information that appears in the *DatabaseName* and *Params* properties.
   - If there was no original connection component, you must transfer the information associated with the BDE alias or that appeared in the dataset's *ConnectionString* property.

   You may want to save this set of parameters under a new connection name. For more details on this process, see "Controlling connections" on page 17-2.

Updating data in dbExpress applications

dbExpress applications use client datasets to support editing. When you post edits to a client dataset, the changes are written to the client dataset's in-memory snapshot of the data, but are not automatically written to the database server. If your original application used a client dataset for caching updates, then you do not need to change anything to support editing on Linux. However, if you relied on the default behavior of most datasets on Windows, which is to write edits to the database server when you post records, you must make changes to accommodate the use of a client dataset.

There are two ways to convert an application that did not previously cache updates:

- You can mimic the behavior of the dataset on Windows by writing code to apply each updated record to the database server as soon as it is posted. To do this, supply the client dataset with an *AfterPost* event handler that applies update to the database server:

```
procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
  with DataSet as TClientDataSet do
    ApplyUpdates(1);
end;
```

- You can adjust your user interface to deal with cached updates. This approach has certain advantages, such as reducing the amount of network traffic and minimizing transaction times. However, if you switch to using cached updates, you must decide when to apply those updates back to the database server, and probably make user interface changes to let users initiate the application of updates or inform provide them with feedback about whether their edits have been written to the database. Further, because update errors are not detected when the user posts a record, you will need to change the way you report such errors to the user, so that they can see which update caused a problem as well as what type of problem occurred.

If your original application used the support provided by the BDE or ADO for caching updates, you will need to make some adjustments in your code to switch to using a client dataset. The following table lists the properties, events, and methods that support cached updates on BDE and ADO datasets, and the corresponding properties, methods and events on *TClientDataSet*:

Table 10.11 Properties, methods, and events for cached updates

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
CachedUpdates	*LockType*	Not needed, client datasets always cache updates.	Determines whether cached updates are in effect.
Not supported.	*CursorType*	Not supported.	Specifies how isolated the dataset is from changes on the server.
UpdatesPending	Not supported.	*ChangeCount*	Indicates whether the local cache contains updated records that need to be applied to the database.
UpdateRecordTypes	*FilterGroup*	*StatusFilter*	Indicates the kind of updated records to make visible when applying cached updates.
UpdateStatus	*RecordStatus*	*UpdateStatus*	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError	Not supported.	*OnReconcileError*	An event for handling update errors on a record-by-record basis.
ApplyUpdates (on dataset or database)	UpdateBatch	*ApplyUpdates*	Applies records in the local cache to the database.

Table 10.11 Properties, methods, and events for cached updates (continued)

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
CancelUpdates	CancelUpdates or CancelBatch	*CancelUpdates*	Removes pending updates from the local cache without applying them.
CommitUpdates	Handled automatically	*Reconcile*	Clears the update cache following successful application of updates.
FetchAll	Not supported	*GetNextPacket* (and *PacketRecords*)	Copies database records to the local cache for editing and updating.
RevertRecord	CancelBatch	*RevertRecord*	Undoes updates to the current record if updates are not yet applied.

Cross-platform Internet applications

An Internet application is a client/server application that uses standard Internet protocols for connecting the client to the server. Because your applications use standard Internet protocols for client/server communications, you can make your applications cross-platform. For example, a server-side program for an Internet application communicates with the client through the Web server software for the machine. The server application is typically written for Linux or Windows, but can also be cross-platform. The clients can be on either platform.

You can use Delphi or Kylix to create Web server applications as CGI or Apache applications for deployment on Linux. On Windows, you can create other types of Web servers such as Microsoft Server DLLs (ISAPI), Netscape Server DLLs (NSAPI), and Windows CGI applications. Only straight CGI applications and some applications that use Web Broker will run on both Windows and Linux.

Porting Internet applications to Linux

If you have existing Internet applications that you want to make cross-platform, you should consider whether you want to port your Web server application or if you want to create a new application on Linux. See Chapter 27, “Creating Internet applications” for information on writing Web servers. If your application uses Web Broker and writes to the Web Broker interface and does not use native API calls, it will not be as difficult to port it to Linux.

If your application writes to ISAPI, NSAPI, Windows CGI, or other Web APIs, it will be more difficult to port. You will need to search through your source files and translate these API calls into Apache (see httpd.pas in the Internet directory for function prototypes for Apache APIs) or CGI calls. You also need to make all other suggested changes described in “Porting VCL applications to CLX” on page 10-2.

Chapter

11

Working with packages and components

A *package* is a special dynamic-link library used by Delphi applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland package library) extension.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used Delphi components reside in a package called vcl. Each time you create an application, it automatically uses vcl. When you compile an application created this way, the application's executable image contains only the code and data unique to it; the common code is in the runtime package called vcl60.bpl. A computer with several package-enabled applications installed on it needs only a single copy of vcl60.bpl, which is shared by all the applications and the Delphi IDE itself.

Delphi ships with several precompiled runtime packages that encapsulate VCL and CLX components. Delphi also uses design-time packages to manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write Delphi components, you can compile your components into design-time packages before installing them.

Why use packages?

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Delphi itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller—saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

Packages and standard DLLs

Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL when you want to build a library that can be called from any application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages:

Table 11.1 Compiled package files

File extension	Contents
dpk	The source file listing the units contained in the package.
dcp	A binary image containing a package header and the concatenation of all dcu files in the package, including all symbol information required by the compiler. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file. You must have a .dcp file to build an application with packages.
dcu	A binary image for a unit file contained in a package. One dcu is created, when necessary, for each unit file.
bpl	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the bpl is the base name of the dpk source file.

You can include VCL or CLX or both types of components in a package. Packages meant to be cross-platform should include CLX components only.

Note Packages share their global data with other modules in an application.

For more information about DLLs and packages, see the *Object Pascal Language Guide*.

Runtime packages

Runtime packages are deployed with Delphi applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's executable file and all the packages (.bpl files) that the application uses.

The .bpl files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required .bpls.

Using packages in an application

To use packages in an application,

1. Load or create a project in the IDE.
2. Choose Project|Options.
3. Choose the Packages tab.
4. Select the "Build with Runtime Packages" check box, and enter one or more package names in the edit box underneath. (Runtime packages associated with installed design-time packages are already listed in the edit box.)
5. To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog. To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog.

   If you edit the Search Path edit box in the Add Runtime Package dialog, you will be changing Delphi's global Library Path.

   You do not need to include file extensions with package names (or the number representing the Delphi release); that is, vcl60.bpl is written as vcl. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with semicolons. For example:

   ```
   rtl;vcl;vcldb;vclado;vclx;Vclbde;
   ```

Packages listed in the Runtime Packages edit box are automatically linked to your application when you compile. Duplicate package names are ignored, and if the edit box is empty the application is compiled without packages.

Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the "Defaults" check box at the bottom of the dialog.

Note When you create an application with packages, you still need to include the names of the original Delphi units in the **uses** clause of your source files. For example, the source file for your main form might begin like this:

```
unit MainForm;

interface

uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
```

The units referenced in this example are contained in the vcl and rtl packages. Nonetheless, you must keep these references in the **uses** clause, even if you use vcl and rtl in your application, or you will get compiler errors. In generated source files, Delphi adds these units to the **uses** clause automatically.

Dynamically loading packages

To load a package at runtime, call the *LoadPackage* function. *LoadPackage* loads the package, checks for duplicate units, and calls the initialization blocks of all units contained in the package. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

Deciding which runtime packages to use

Delphi ships with several precompiled runtime packages, including rtl and vcl, which supply basic language and component support.

The vcl package contains the most commonly used components; the rtl package includes all the non-component system functions and Windows interface elements. It does not include database or other special components, which are available in separate packages.

To create a client/server database application that uses packages, you need at least three runtime packages: vcl and vcldb. If you want to use Outline components in your application, you also need vclx. To use these packages, choose Project|Options, select the Packages tab, and enter the following list in the Runtime Packages edit box.

```
rtl;vcl;Vcldb;vclx;
```

Actually, you don't have to include vcl and rtl, because they are referenced in the Requires clause of vcldb. (See "Requires clause" on page 11-8.) Your application will compile just the same whether or not vcl and rtl are included in the Runtime Packages edit box.

Custom packages

A custom package is either a bpl you code and compile yourself or a precompiled package from a third-party vendor. To use a custom runtime package with an application, choose Project|Options and add the name of the package to the Runtime Packages edit box on the Packages page. For example, suppose you have a statistical package called stats.bpl. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:

```
rtl;vcl;vcldb;stats
```

If you create your own packages, you can add them to the list as needed.

Design-time packages

Design-time packages are used to install components on the IDE's Component palette and to create special property editors for custom components.

Delphi ships with many design-time component packages preinstalled in the IDE. Which ones are installed depends on which version of Delphi you are using and whether or not you have customized it. You can view a list of what packages are installed on your system by choosing Component | Install Packages.

The design-time packages work by calling runtime packages, which they reference in their Requires clauses. (See "Requires clause" on page 11-8.) For example, dclstd references vcl. Dclstd itself contains additional functionality that makes most of the standard components available on the Component palette.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The dclusr design-time package is provided as a default container for new components.

Installing component packages

All components are installed in the IDE as packages. If you've written your own components, create and compile a package that contains them. (See "Creating and editing packages" on page 11-6.) Your component source code must follow the model described in Part V, "Creating custom components".

To install or uninstall your own components, or components from a third-party vendor, follow these steps:

1. If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with .bpl, .dcp, and .dcu files, be sure to copy all of them. (For information about these files, see , "Package files created by a successful compilation.")

   The directory where you store the .dcp file—and the .dcu files, if they are included with the distribution—must be in the Delphi Library Path.

   If the package is shipped as a .dpc (package collection) file, only the one file needs to be copied; the .dpc file contains the other files. (For more information about package collection files, see "Package collection files" on page 11-13.)

2. Choose Component | Install Packages from the IDE menu, or choose Project | Options and click the Packages tab.

3. A list of available packages appears under "Design packages."

   - To install a package in the IDE, select the check box next to it.
   - To uninstall a package, deselect its check box.
   - To see a list of components included in an installed package, select the package and click Components.

- To add a package to the list, click Add and browse in the Open Package dialog box for the directory where the .bpl or .dpc file resides (see step 1). Select the .bpl or .dpc file and click Open. If you select a .dpc file, a new dialog box appears to handle the extraction of the .bpl and other files from the package collection.
- To remove a package from the list, select the package and click Remove.

4 Click OK.

The components in the package are installed on the Component palette pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the default settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the Packages tab of the Project Options dialog box.

To remove components from the Component palette without uninstalling a package, select Component | Configure Palette, or select Tools | Environment Options and click the Palette tab. The Palette options tab lists each installed component along with the name of the Component palette page where it appears. Selecting any component and clicking Hide removes the component from the palette.

Creating and editing packages

Creating a package involves specifying

- A *name* for the package.
- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which contain the functionality of the compiled .bpl. The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

Package source files, which end with the .dpk extension, are generated by the Package editor.

Creating a package

To create a package, follow the procedure below. Refer to "Understanding the structure of a package" on page 11-8 for more information about the steps outlined here.

Note Do not use IFDEFs in a package file (.dpk) such as when doing cross-platform development. You can use them in the source code, however.

1 Choose File | New, select the Package icon, and click OK.

2. The generated package is displayed in the Package editor.
3. The Package editor shows a *Requires* node and a *Contains* node for the new package.
4. To add a unit to the **contains** clause, click the Add to package speed button. In the Add unit page, type a .pas file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the Package editor. You can add additional units by repeating this step.
5. To add a package to the **requires** clause, click the Add to package speed button. In the Requires page, type a .dcp file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the Package editor. You can add additional packages by repeating this step.
6. Click the Options speed button, and decide what kind of package you want to build.
   - To create a design-time only package (a package that cannot be used at runtime), select the Designtime only radio button. (Or add the {$DESIGNONLY} compiler directive to the dpk file.)
   - To create a runtime-only package (a package that cannot be installed), select the Runtime only radio button. (Or add the {$RUNONLY} compiler directive to the dpk file.)
   - To create a package that is available at both design time and runtime, select the Designtime and runtime radio button.
7. In the Package editor, click the Compile package speed button to compile your package.

Editing an existing package

You can open an existing package for editing in several ways:

- Choose File | Open (or File | Reopen) and select a dpk file.
- Choose Component | Install Packages, select a package from the Design Packages list, and click the Edit button.
- When the Package editor is open, select one of the packages in the Requires node, right-click, and choose Open.

To edit a package's description or set usage options, click the Options speed button in the Package editor and select the Description tab.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the defproj.dof file.

Editing package source files manually

Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi package) extension to avoid confusion with other files containing Object Pascal source code.

To open a package source file in the Code editor,

1. Open the package in the Package editor.
2. Right-click in the Package editor and select View Source.
   - The **package** heading specifies the name for the package.
   - The **requires** clause lists other, external packages used by the current package. If a package does not contain any units that use units in another package, then it doesn't need a **requires** clause.
   - The **contains** clause identifies the unit files to be compiled and bound into the package. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains clause (the compiler will give a warning).

   For example, the following code declares the vcldb package (in the source file vcldb60.bpl):

```
package vcldb;
   requires vcldb;
   contains rtl, vcl, Db, DBActns, DBOleCtl, Dbcgrids, dbCommon, dbConsts, Dbctrls,
Dbgrids, Dblogdlg, SQLTimSt, FmtBcd;
end.
```

Understanding the structure of a package

Packages include the following parts:

- Package name
- Requires clause
- Contains clause

Naming packages

Package names must be unique within a project. If you name a package STATS, the Package editor generates a source file for it called STATS.dpk; the compiler generates an executable and a binary image called STATS.bpl and STATS.dcp, respectively. Use STATS to refer to the package in the **requires** clause of another package, or when using the package in an application.

Requires clause

The **requires** clause specifies other, external packages that are used by the current package. An external package included in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's **requires** clause or you should add them. If the other packages are omitted from the **requires** clause, the compiler will import them into your package 'implicitly contained units'.

Note Most packages that you create will require rtl. If using VCL components, you'll also need to include the vcl package. If using CLX components for cross-platform programming, you need to include VisualCLX.

Avoiding circular package references

Packages cannot contain circular references in their **requires** clause. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Handling duplicate package references

Duplicate references in a package's **requires** clause—or in the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

Contains clause

The **contains** clause identifies the unit files to be bound into the package. If you are writing your own package, put your source code in pas files and include them in the **contains** clause.

Avoiding redundant source code uses

A package cannot appear in the **contains** clause of another package.

All units included directly in a package's **contains** clause, or included indirectly in any of those units, are bound into the package at compile time.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, *including the Delphi IDE*. This means that if you create a package that contains one of the units in vcl you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's **requires** clause.

Compiling packages

You can compile a package from the IDE or from the command line. To recompile a package by itself from the IDE,

1. Choose File | Open.
2. Select Delphi package (*.dpk) from the Files of Type drop-down list.
3. Select a .dpk file in the dialog.
4. When the Package editor opens, click the Compile speed button.

You can insert compiler directives into your package source code. For more information, see "Package-specific compiler directives", below.

If you compile from the command line, several package-specific switches are available. For more information, see "Using the command-line compiler and linker" on page 11-12.

Package-specific compiler directives

The following table lists package-specific compiler directives that you can insert into your source code.

Table 11.2 Package-specific compiler directives

Directive	Purpose
{$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{$G-} or {IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit "weakly." See "Weak packaging" on page 11-11 below.
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

Note Including **{$DENYPACKAGEUNIT ON}** in your source code prevents the unit file from being packaged. Including **{$G-}** or **{$IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the IDE. Other compiler directives may be included, if appropriate, in package source code. See Compiler directives in the online help for information on compiler directives not discussed here.

Refer to "Creating packages and DLLs" on page 5-9 for additional directives that can be used in all libraries.

Weak packaging

The **$WEAKPACKAGEUNIT** directive affects the way a .dcu file is stored in a package's .dcp and .bpl files. (For information about files generated by the compiler, see "Package files created by a successful compilation" on page 11-12.) If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged."

For example, suppose you've created a package called PACK that contains only one unit, UNIT1. Suppose UNIT1 does not use any further units, but it makes calls to RARE.dll. If you put **{$WEAKPACKAGEUNIT ON}** in UNIT1.pas when you compile your package, UNIT1 will not be included in PACK.bpl; you will not have to distribute copies of RARE.dll with PACK. However, UNIT1 will still be included in PACK.dcp. If UNIT1 is referenced by another package or application that uses PACK, it will be copied from PACK.dcp and compiled directly into the project.

Now suppose you add a second unit, UNIT2, to PACK. Suppose that UNIT2 uses UNIT1. This time, even if you compile PACK with **{$WEAKPACKAGEUNIT ON}** in UNIT1.pas, the compiler will include UNIT1 in PACK.bpl. But other packages or applications that reference UNIT1 will use the (non-packaged) copy taken from PACK.dcp.

Note Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **$WEAKPACKAGEUNIT** directive is an advanced feature intended for developers who distribute their packages to other Delphi programmers. It can help you to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, Delphi's PenWin unit references PenWin.dll. Most projects don't use PenWin, and most computers don't have PenWin.dll installed on them. For this reason, the PenWin unit is weakly packaged in vcl. When you compile a project that uses PenWin and the vcl package, PenWin is copied from VCL60.dcp and bound directly into your project; the resulting executable is statically linked to PenWin.dll.

If PenWin were not weakly packaged, two problems would arise. First, vcl itself would be statically linked to PenWin.dll, and so you could not load it on any computer which didn't have PenWin.dll installed. Second, if you tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both vcl and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of vcl.

Using the command-line compiler and linker

When you compile from the command line, you can use the package-specific switches listed in the following table.

Table 11.3 Package-specific command-line compiler switches

Switch	Purpose
-$G-	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
-LE*path*	Specifies the directory where the package bpl file will be placed.
-LN*path*	Specifies the directory where the package dcp file will be placed.
-LU*package*	Use packages.
-Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Note Using the **-$G-** switch may prevent a package from being used in the same application with other packages. Other command-line options may be used, if appropriate, when compiling packages. See “The Command-line compiler” in the online help for information on command-line options not discussed here.

Package files created by a successful compilation

To create a package, you compile a source file that has a .dpk extension. The base name of the .dpk file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called traypak.dpk, the compiler creates a package called traypak.bpl.

The following table lists the files produced by the successful compilation of a package.

Table 11.4 Compiled package files

File extension	Contents
dcp	A binary image containing a package header and the concatenation of all dcu files in the package. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file.
dcu	A binary image for a unit file contained in a package. One dcu is created, when necessary, for each unit file.
bpl	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the bpl is the base name of the dpk source file.

When compiled, the bpi, bpl, and lib files are generated by default in the directories specified in Library page of the Tools | Environment Options dialog. You can override the default settings by clicking the Options speed button in the Package editor to display the Project Options dialog; make any changes on the Directories/ Conditionals page.

Deploying packages

You deploy packages much like you deploy other applications. For general deployment information, refer to Chapter 13, “Deploying applications”.

Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application’s .exe file as well as all the library (.bpl or .dll) files that the application calls. If the library files are in a different directory from the .exe file, they must be accessible through the user’s Path. You may want to follow the convention of putting library files in the Windows\System directory. If you use InstallShield Express, your installation script can check the user’s system for any packages it requires before blindly reinstalling them.

Distributing packages to other developers

If you distribute runtime or design-time packages to other Delphi developers, be sure to supply both .dcp and .bpl files. You will probably want to include .dcu files as well.

Package collection files

Package collections (.dpc files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including bpls and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .pce container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.

To create a package collection,

1. Choose Tools | Package Collection Editor to open the Package Collection editor.
2. Click the Add a Package speed button, then select a bpl in the Select Package dialog and click Open. To add more bpls to the collection, click the Add a Package speed button again. A tree diagram on the left side of the Package editor displays the bpls as you add them. To remove a package, select it and click the Remove Package speed button.
3. Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields will appear:
    - In the Author/Vendor Name edit box, you can enter optional information about your package collection that will appear in the Installation dialog when users install packages.

- Under Directory List, list the default directories where you want the files in your package collection to be installed. Use the Add, Edit, and Delete buttons to edit this list. For example, suppose you want all source code files to be copied to the same directory. In this case, you might enter `Source` as a Directory Name with `C:\MyPackage\Source` as the Suggested Path. The Installation dialog box will display `C:\MyPackage\Source` as the suggested path for the directory.

4 In addition to bpls, your package collection can contain .dcp, .dcu, and .pas (unit) files, documentation, and any other files you want to include with the distribution. Ancillary files are placed in file groups associated with specific packages (bpls); the files in a group are installed only when their associated bpl is installed. To place ancillary files in your package collection, select a bpl in the tree diagram and click the Add File Group speed button; type a name for the file group. Add more file groups, if desired, in the same way. When you select a file group, new fields will appear on the right in the Package Collection editor,

- In the Install Directory list box, select the directory where you want files in this group to be installed. The drop-down list includes the directories you entered under Directory List in step 3, above.
- Check the Optional Group check box if you want installation of the files in this group to be optional.
- Under Include Files, list the files you want to include in this group. Use the Add, Delete, and Auto buttons to edit the list. The Auto button allows you to select all files with specified extensions that are listed in the **contains** clause of the package; the Package Collection editor uses Delphi's global Library Path to search for these files.

5 You can select installation directories for the packages listed in the **requires** clause of any package in your collection. When you select a bpl in the tree diagram, four new fields appear on the right side of the Package Collection editor:

- In the Required Executables list box, select the directory where you want the .bpl files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection Editor searches for these files using Delphi's global Library Path and lists them under Required Executable Files.
- In the Required Libraries list box, select the directory where you want the .dcp files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection Editor searches for these files using Delphi's global Library Path and lists them under Required Library Files.

6 To save your package collection source file, choose File | Save. Package collection source files should be saved with the .pce extension.

7 To build your package collection, click the Compile speed button. The Package Collection editor generates a .dpc file with the same name as your source (.pce) file. If you have not yet saved the source file, the editor queries you for a file name before compiling.

To edit or recompile an existing .pce file, select File | Open in the Package Collection editor and locate the file you want to work with.

Chapter

12

Creating international applications

This chapter discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

Internationalization and localization

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization
- Localization

If your version of Delphi includes the Translation Tools, you can use the them to manage localization. For more information, see the online Help for the Translation Tools (ETM.hlp).

Internationalization

Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Windows supports a large set of locales, each of which is described by a language and country pair.

Localization

Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified to be aware of the different tax laws in different countries.

Internationalizing applications

You need to complete the following steps to create internationalized applications:

- You must enable your code to handle strings from international character sets.
- You need to design your user interface so that it can accommodate the changes that result from localization.
- You need to isolate all resources that need to be localized.

Enabling application code

You must make sure that the code in your application can handle the strings it will encounter in the various target locales.

Character sets

The United States edition of Windows uses the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-JIS character set (code page 932), which represents Japanese characters as multibyte character codes.

There are generally three types of characters sets:

- Single-byte
- Multibyte
- Fixed-width multibyte

Windows and Linux both support single-byte and multibyte character sets as well as Unicode. With a single-byte character set, each byte in a string represents one character. The ANSI character set used by many Western operating systems is a single-byte character set.

In a multibyte character set, some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages, while the UTF-8 character set used by Linux is a multibyte encoding of Unicode.

OEM and ANSI character sets

It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).

Multibyte character sets

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) *char* type. These languages have too many characters to be represented using the 1-byte *char*. Instead, a multibyte string can contain one or more bytes per character. AnsiStrings can contain a mix of single-byte and multibyte characters.

The lead byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent bytes can sometimes be the same as the character code for a separate 1-byte character, or it can fall in the range reserved for the first byte of multibyte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or is part of a multibyte character is to read the string, starting at the beginning, parsing it into 2 or more byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into multibyte characters. Delphi provides you with many runtime library functions that allow you to do this, many of which are listed here:

AdjustLineBreaks
AnsiCompareFileName
AnsiExtractQuotedStr
AnsiLastChar
AnsiLowerCase
AnsiLowerCaseFileName
AnsiPos
AnsiQuotedStr
AnsiStrComp
AnsiStrIComp
AnsiStrLastChar
AnsiStrLComp
AnsiStrLIComp
AnsiStrLower
AnsiStrPos
AnsiStrRScan
AnsiStrScan
AnsiStrUpper
AnsiUpperCase
AnsiUpperCaseFileName
ByteToCharIndex
ByteToCharLen
ByteType
ChangeFileExt
CharToByteIndex
CharToByteLen
ExtractFileDir
ExtractFileExt
ExtractFileName
ExtractFilePath
ExtractRelativePath
FileSearch
IsDelimiter
IsPathDelimiter
LastDelimiter
StrByteType
StringReplace
WrapText

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.

Wide characters

Another approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. Unicode characters and strings are also called wide characters and wide character strings. In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words.

The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi/Kylix supports UCS-2 on both platforms. Because wide characters are two bytes instead of one, the character set can represent many more different characters.

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

The biggest disadvantage of working with wide characters is that Windows 9x only supports a few wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require additional code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

Including bi-directional functionality in applications

Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional.

TApplication has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties. The following table lists VCL objects that have these properties:

Table 12.1 VCL objects that support BiDi

Component palette page	VCL object
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel

Table 12.1 VCL objects that support BiDi (continued)

Component palette page	VCL object
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar
Additional	TActionMainMenuBar
	TActionToolBar
	TBitBtn
	TCheckListBox
	TColorBox
	TDrawGrid
	TLabeledEdit
	TMaskEdit
	TScrollBox
	TSpeedButton
	TStaticLabel
	TStaticText
	TStringGrid
	TValueListEditor
Win32	TComboBoxEx
	TDateTimePicker
	THeaderControl
	THotKey
	TListView
	TMonthCalendar
	TPageControl
	TRichEdit
	TStatusBar
	TTreeView
Data Controls	TDBCheckBox
	TDBComboBox
	TDBEdit
	TDBGrid
	TDBListBox
	TDBLookupComboBox
	TDBLookupListBox
	TDBMemo

Table 12.1 VCL objects that support BiDi (continued)

Component palette page	VCL object
	TDBRadioGroup
	TDBRichEdit
	TDBText
QReport	TQRDBText
	TQRExpr
	TQRLabel
	TQRMemo
	TQRPreview
	TQRSysData
Other classes	TApplication (has no *ParentBiDiMode*)
	TBoundLabel
	TControl (has no *ParentBiDiMode*)
	TCustomHeaderControl (has no *ParentBiDiMode*)
	TForm
	TFrame
	THeaderSection
	THintWindow (has no *ParentBiDiMode*)
	TMenu
	TStatusPanel
	TTabControl
	TValueListEditor

Notes *THintWindow* picks up the *BiDiMode* of the control that activated the hint.

Bi-directional properties

The objects listed in Table 12.1, "VCL objects that support BiDi," on page 12-4 have the properties *BiDiMode* and *ParentBiDiMode*. These properties, along with *TApplication*'s *BiDiKeyboard* and *NonBiDiKeyboard*, support bi-directional localization.

Note Bi-directional properties are not available in CLX for cross-platform programming.

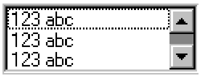
BiDiMode property

The property *BiDiMode* is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly.*

bdLeftToRight

bdLeftToRight draws text using left to right reading order, and the alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

Figure 12.1 TListBox set to bdLeftToRight

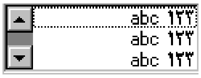


123 abc
123 abc
123 abc

bdRightToLeft

bdRightToLeft draws text using right to let reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left-to-right.

Figure 12.2 TListBox set to bdRightToLeft

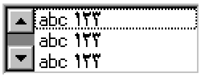


abc ١٢٣
abc ١٢٣
abc ١٢٣

bdRightToLeftNoAlign

bdRightToLeftNoAlign draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

Figure 12.3 TListBox set to bdRightToLeftNoAlign

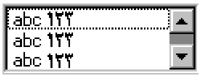


abc ١٢٣
abc ١٢٣
abc ١٢٣

bdRightToLeftReadingOnly

bdRightToLeftReadingOnly draws text using right to left reading order, and the alignment and scroll bars are not changed.

Figure 12.4 TListBox set to bdRightToLeftReadingOnly



abc ١٢٣
abc ١٢٣
abc ١٢٣

ParentBiDiMode property

ParentBiDiMode is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from *Application*. If all the *ParentBiDiMode* properties are *True*, when you change *Application's BiDiMode* property, all forms and controls in the project are updated with the new setting.

FlipChildren method

The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupBox*. *FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped.

Delphi flips the controls by changing the Left property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control, after it is flipped the edit control's right side is five pixels from the right edge of the parent control. If the edit control is left aligned, calling *FlipChildren* will make the control right aligned.

To flip a control at design-time select Edit|Flip Children and select either All or Selected, depending on whether you want to flip all the controls, or just the children of the selected control. You can also flip a control by selecting the control on the form, right-clicking, and selecting Flip Children from the context menu.

Note Selecting an edit control and issuing a Flip Children|Selected command does nothing. This is because edit controls are not containers.

Additional methods

There are several other methods useful for developing applications for bi-directional users.

Method	Description
OkToChangeFieldAlignment	Used with database controls. Checks to see if the alignment of a control can be changed.
DBUseRightToLeftAlignment	A wrapper for database controls for checking alignment.
ChangeBiDiModeAlignment	Changes the alignment parameter passed to it. No check is done for *BiDiMode* setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone.
IsRightToLeft	Returns *True* if any of the right to left options are selected. If it returns *False* the control is in left to right mode.
UseRightToLeftReading	Returns *True* if the control is using right to left reading.
UseRightToLeftAlignment	Returns *True* if the control is using right to left alignment. It can be overridden for customization.
UseRightToLeftScrollBar	Returns *True* if the control is using a left scroll bar.
DrawTextBiDiModeFlags	Returns the correct draw text flags for the BiDiMode of the control.
DrawTextBiDiModeFlagsReadingOnly	Returns the correct draw text flags for the *BiDiMode* of the control, limiting the flag to its reading order.
AddBiDiModeExStyle	Adds the appropriate *ExStyle* flags to the control that is being created.

Locale-specific features

You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings.

VCL components offer support in programming the IME. Most windowed controls that work directly with text input have an *ImeName* property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an *ImeMode* property that specifies how the IME should convert keyboard input. *TWinControl* introduces several protected methods that you can use to control the IME from classes you define. In addition, the global *Screen* variable provides information about the IMEs available on the user's system.

The global *Screen* variable (available in VCL and CLX) also provides information about the keyboard mapping installed on the user's system. You can use this to obtain locale-specific information about the environment in which your application is running.

Designing the user interface

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. Table 12.2 provides a rough estimate of how much expansion you should plan for given the length of your English strings:

Table 12.2 Estimating string lengths

Length of English string (in characters)	Expected increase
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

Graphic images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if your application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

Formats and sort order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, if you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, 2-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

Keyboard mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

Isolating resources

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Delphi automatically creates a .dfm (.xfm in CLX applications) file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the Object Pascal Language Guide. It is best to include all resource strings in a single, separate unit.

Creating resource DLLs

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL.

Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the project, and generate a project for a resource only DLL that contains the relevant forms and the created RES file. The RES file is compiled from the new RC file.

You should create a resource DLL for each translation you want to support. Each resource DLL should have a file name extension specific to the target locale. The first two characters indicate the target language, and the third character indicates the country of the locale. If you use the Resource DLL wizard, this is handled for you. Otherwise, use the following code to obtain the locale code for the target translation:

```
unit locales;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleinfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;

{ Called for each supported locale. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
```

```
    with Languages do
    begin
      for I := 0 to Count - 1 do
      begin
        ListBox1.Items.Add(Name[I]);
      end;
    end;
  end;
```

Using resource DLLs

The executable, DLLs, and packages that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your EXE, DLL, or BPL files.

When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, DLL, or package. If there is not a resource module that matches both the language and the country, your application will try to locate a resource module that matches just the language. If there is no resource module that matches the language, your application will use the resources compiled with the executable, DLL, or package.

If you want your application to use a different resource module than the one that matches the locale of the local system, you can set a locale override entry in the Windows registry. Under the HKEY_CURRENT_USER\Software\Borland\Locales key, add your application's path and file name as a string value and set the data value to the extension of your resource DLLs. At startup, the application will look for resource DLLs with this extension before trying the system locale. Setting this registry entry allows you to test localized versions of your application without changing the locale on your system.

For example, the following procedure can be used in an install or setup program to set the registry key value that indicates the locale to use when loading Delphi applications:

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software\Borland\Locales', True) then
      Reg.WriteString(LocalOverride, FileName);
  finally
    Reg.Free;
end;
```

Within your application, use the global *FindResourceHInstance* function to obtain the handle of the current resource module. For example:

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
```

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource DLLs.

Dynamic switching of resource DLLs

In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the ReInit unit in your **uses** statement. (ReInit is located in the Richedit sample in the Demos directory.) To switch languages, you should call *LoadResourceModule*, passing the LCID for the new language, and then call *ReinitializeForms*.

For example, the following code switches the interface language to French:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

The advantage of this technique is that the current instance of the application and all of its forms are used. It is not necessary to update the registry settings and restart the application or reacquire resources required by the application, such as logging in to database servers.

When you switch resource DLLs the properties specified in the new DLL overwrite the properties in the running instances of the forms.

Note Any changes made to the form properties at runtime will be lost. Once the new DLL is loaded, default values are not reset. Avoid code that assumes that the form objects are reinitialized to the their startup state, apart from differences due to localization.

Localizing applications

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

Localizing resources

Ideally, your resources have been isolated into a resource DLL that contains form files (.dfm or .xfm) and a resource file. You can open your forms in the IDE and translate the relevant properties.

Note In a resource DLL project, you cannot add or delete components. It is possible, however, to change properties in ways that could cause runtime errors, so be careful to modify only those properties that require translation. To avoid mistakes, you can configure the Object Inspector to display only localizable properties; to do so, right-click in the Object Inspector and use the View menu to filter out unwanted property categories.

You can open the RC file and translate relevant strings. Use the StringTable editor by opening the RC file from the Project Manager.

Chapter

13

Deploying applications

Once your Delphi application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. What is required by a given application varies, depending on the type of application. The following sections describe considerations when deploying different types of applications:

- Deploying general applications
- Deploying CLX applications
- Deploying database applications
- Deploying Web applications
- Programming for varying host environments
- Software license requirements

Note Information included in these sections is for deploying applications on Windows. If writing cross-platform applications for deployment on Linux, you need to refer to deployment information provided in your Kylix documentation.

Deploying general applications

Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. These are the main deployment concerns common to nearly all types of applications:

- Using installation programs
- Identifying application files

Delphi applications that access databases and those that run across the Web require additional installation steps beyond those that apply to general applications. For additional information on installing database applications, see “Deploying database applications” on page 13-6. For more information on installing Web applications, see “Deploying Web applications” on page 13-9. For more information on installing ActiveX controls, see “Deploying an ActiveX control on the Web” on page 38-15.

Using installation programs

Simple Delphi applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for BDE database applications.

InstallShield Express is a setup toolkit that is bundled with Delphi. InstallShield Express is certified for use with Delphi and the Borland Database Engine. It is based on Windows Installer (MSI) technology.

InstallShield Express is not automatically installed when Delphi is installed, so it must be manually installed if you want to use it to create installation programs. Run the installation program from the Delphi CD to install InstallShield Express. For more information on using InstallShield Express to create installation programs, see the InstallShield Express online help.

Other setup toolkits are available. However, if deploying BDE database applications, you should only use toolkits based on MSI technology and those which are certified to deploy the Borland Database Engine.

Identifying application files

Besides the executable file, a number of other files may need to be distributed with an application.

- Application files
- Package files
- Merge modules
- ActiveX controls

Application files

The following types of files may need to be distributed with an application.

Table 13.1 Application files

Type	File name extension
Program files	.exe and .dll
Package files	.bpl and .dcp
Help files	.hlp, .cnt, and .toc (if used) or any other help files your application supports
ActiveX files	.ocx (sometimes supported by a DLL)
Local table files	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

Package files

If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. You can also use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express. See the next section for details.

Borland recommends installing the runtime package files supplied by Borland in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you created, it is recommended that you install them in the same directory as the application. Only the .BPL files need to be distributed.

Note If deploying packages with CLX applications, you need to include clx60.bpl rather than vcl60.bpl.

If you are distributing packages to other developers, supply both the .BPL and the .DCP files.

Merge modules

InstallShield Express 3.0 is based on Windows Installer (MSI) technology. That is why Delphi includes merge modules. Merge modules provide a standard method that you can use to deliver shared code, files, resources, Registry entries, and setup logic to applications as a single compound file. You can use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express.

The runtime libraries have some interdependencies because of the way they are grouped together. The result of this is that when one package is added to an install project, the install tool will automatically add or report a dependency on one or more other packages.For example, if you add the VCLInternet merge module to an install project, the install tool should also automatically add or report a dependency on the VCLDatabase and StandardVCL modules.

The dependencies for each merge module are listed in the table below. The various install tools may react to these dependencies differently. The InstallShield for Windows Installer automatically adds the required modules if it can find them.

Other tools may simply report a dependency or may generate a build failure if all required modules are not included in the project.

Table 13.2 Merge modules and their dependencies

Merge module	BPLs included	Dependencies
ADORTL	adortl60.bpl	DatabaseRTL, BaseRTL
BaseClientDataSet	cds60.bpl	DatabaseRTL, BaseRTL, DataSnap, dbExpress
BaseRTL	rtl60.bpl	No dependencies
BaseVCL	vcl60.bpl, vclx60.bpl	BaseRTL
BDEClientDataSet	bdecds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress, BDERTL
BDEInternet	inetdbbde60.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl60.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl60.bpl	BaseRTL
DatabaseVCL	vcldb60.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsnap60.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsnapcon60.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapCorba	dsnapcrba60.bpl	DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DataSnapEntera	dsnapent60.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx60.bpl	DatabaseVCL, BaseVCL, BaseRTL
dbExpress	dbexpress60.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress60.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss60.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
FastNet	nmfast60.bpl	BaseVCL, BaseRTL
InterbaseVCL	vclib60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress, BaseVCL
Internet	inet60.bpl, inetdb60.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy60.bpl	BaseVCL, BaseRTL
Office2000Components	dcloffice2k60.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
QuickReport	qrpt60.bpl	BaseVCL, BaseRTL, BDERTL, DatabaseRTL
SampleVCL	vclsmp60.bpl	BaseVCL, BaseRTL
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	BaseVCL, BaseRTL
VCLIE	vclie60.bpl	BaseVCL, BaseRTL
VisualCLX	visualclx60.bpl	BaseRTL
WebDataSnap	webdsnap60.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL

Table 13.2 Merge modules and their dependencies (continued)

Merge module	BPLs included	Dependencies
WebSnap	websnap60.bpl, vcljpg60.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmlrtl60.bpl	Internet, DatabaseRTL, BaseRTL

ActiveX controls

Certain components bundled with Delphi are ActiveX controls. The component wrapper is linked into the application's executable file (or a runtime package), but the .OCX file for the component also needs to be deployed with the application. These components include

- Chart FX, copyright SoftwareFX Inc.
- VisualSpeller Control, copyright Visual Components, Inc.
- Formula One (spreadsheet), copyright Visual Components, Inc.
- First Impression (VtChart), copyright Visual Components, Inc.
- Graph Custom Control, copyright Bits Per Second Ltd.

ActiveX controls of your own creation need to be registered on the deployment computer before use. Installation programs such as InstallShield Express automate this registration process. To manually register an ActiveX control, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows versions).

DLLs that support an ActiveX control also need to be distributed with an application.

Helper applications

Helper applications are separate programs without which your Delphi application would be partially or completely unable to function. Helper applications may be those supplied with the operating system, by Borland, or they might be third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the documentation for the helper for specific information.

DLL locations

You can install .dll files used only by a single application in the same directory as the application. DLLs that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community DLLs is to place them either in the Windows or the Windows\ System directory. A better way is to create a dedicated directory for the common .dll files, similar to the way the Borland Database Engine is installed.

Deploying CLX applications

If you are writing cross-platform applications that will be deployed on both Windows and Linux, you need to compile and deploy the applications on both platforms. The steps for deploying CLX applications are the same as those for general applications. For information on deploying general applications, see “Deploying general applications” on page 13-1. For information on installing database CLX applications, see “Deploying database applications” on page 13-6.

Note When deploying CLX applications on Windows, you need to include qtintf.dll with the application to include the CLX runtime. If deploying packages with CLX applications, you need to include clx60.bpl rather than vcl60.bpl.

See Chapter 10, “Using CLX for cross-platform development” for information on writing CLX applications.

Deploying database applications

Applications that access databases involve special installation considerations beyond copying the application’s executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application’s executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require even more specialized handling on installation, because the files that make up the application are typically located on multiple computers.

Since several different database technologies (ADO, BDE, dbExpress, and InterBase Express) are supported, deployment requirements differ for each. Regardless of which you are using, you need to make sure that the client side software is installed on the system where you plan to run the database application. BDE, ADO, and dbExpress also require drivers to interact with the client-side software of the database. InterBase does not require drivers because the IBX components communicate directly with the database.

Specific information on how to deploy dbExpress, BDE, and multi-tiered database applications is described in the following sections:

- Deploying dbExpress database applications
- Deploying BDE applications
- Deploying multi-tiered database applications (DataSnap)

Database applications that use client datasets such as *TClientDataSet* or *TSQLClientDataSet* or dataset providers require you to include libmidas.dcu and crtl.dcu (for static linking when providing a standalone executable); if you are packaging your application (with the executable and any needed DLLs), you need to include Midas.dll.

If deploying database applications that use ADO, you need to be sure that MDAC version 2.1 or later is installed on the system where you plan to run the application. MDAC is automatically installed with software such as Windows 2000 and Internet Explorer version 5 or later. You also need to be sure the drivers for the database server you want to connect to are installed on the client. No other deployment steps are required.

If deploying database applications that use InterBase Express, you need to be sure that the InterBase client is installed on the system where you plan to run the application. InterBase requires gd32.dll and interbase.msg to be located in an accessible directory. No other deployment steps are required. InterBase Express components communicate directly with the database and do not require additional drivers. For more information, refer to the Embedded Installation Guide posted on the Borland Web site.

In addition to the technologies described here, you can also use third-party database engines to provide database access for Delphi applications. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

Deploying dbExpress database applications

dbExpress is a set of drivers that provide fast access to database information. dbExpress components support cross-platform development because they are also available on Linux. Refer to Chapter 22, “Using unidirectional datasets” for more information about using the dbExpress components.

You can deploy dbExpress applications either as a stand-alone executable file or as an executable file that includes associated dbExpress driver DLLs.

To deploy dbExpress applications as standalone executable files, the dbExpress object files must be statically linked into your executable. You do this by including the following DCUs, located in the lib directory:

Table 13.3 dbExpress deployment as standalone executable

Database unit	When to include
dbExpInt	Applications connecting to InterBase databases
dbExpOra	Applications connecting to Oracle databases
dbExpDb2	Applications connecting to DB2 databases
dbExpMy	Applications connecting to MySQL databases
Crtl, MidasLib	Required by dbExpress executables that use client datasets such as *TSQLClientDataSet*

If you are not deploying a standalone executable, you can deploy associated dbExpress drivers and DataSnap DLLs with your executable. The following table lists the appropriate DLLs and when to include them:

Table 13.4 dbExpress deployment with driver DLLs

Database DLL	When to deploy
dbexpint.dll	Applications connecting to InterBase databases
dbexpora.dll	Applications connecting to Oracle databases
dbexpdb2.dll.	Applications connecting to DB2 databases
dbexpmy.dll	Applications connecting to MySQL databases
Midas.dll	Required by database applications that use client datasets

Deploying BDE applications

The Borland Database Engine (BDE) defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables.

Database access for an application is provided by various database engines. An application can use the BDE or a third-party database engine. SQL Links is provided (not available in all versions) to enable native access to SQL database systems. The following sections describe installation of the database access elements of an application:

- Borland Database Engine
- SQL Links

Borland Database Engine

For standard Delphi data components to have database access, the Borland Database Engine (BDE) must be present and accessible. See the BDEDEPLOY document for specific rights and limitations on redistributing the BDE.

Borland recommends use of InstallShield Express (or other certified installation program) for installing the BDE. InstallShield Express will create the necessary registry entries and define any aliases the application may require. Using a certified installation program to deploy the BDE files and subsets is important because:

- Improper installation of the BDE or BDE subsets can cause other applications using the BDE to fail. Such applications include not only Borland products, but many third-party programs that use the BDE.
- Under Windows 9x and Windows NT, BDE configuration information is stored in the Windows registry instead of .INI files, as was the case under 16-bit Windows. Making the correct entries and deletions for install and uninstall is a complex task.

It is possible to install only as much of the BDE as an application actually needs. For instance, if an application only uses Paradox tables, it is only necessary to install that portion of the BDE required to access Paradox tables. This reduces the disk space needed for an application. Certified installation programs, like InstallShield Express, are capable of performing partial BDE installations. Be sure to leave BDE system files that are not used by the deployed application, but that are needed by other programs.

SQL Links

SQL Links provides the drivers that connect an application (through the Borland Database Engine) with the client software for an SQL database. See the DEPLOY document for specific rights and limitations on redistributing SQL Links. As is the case with the Borland Database Engine (BDE), SQL Links must be deployed using InstallShield Express (or other certified installation program).

Note SQL Links only connects the BDE to the client software, not to the SQL database itself. It is still necessary to install the client software for the SQL database system used. See the documentation for the SQL database system or consult the vendor that supplies it for more information on installing and configuring client software.

Table 13.5 shows the names of the driver and configuration files SQL Links uses to connect to the different SQL database systems. These files come with SQL Links and are redistributable in accordance with the Delphi license agreement.

Table 13.5 SQL database client software files

Vendor	Redistributable files
Oracle 7	SQLORA32.DLL and SQL_ORA.CNF
Oracle8	SQLORA8.DLL and SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL and SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL and SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL and SQL_MSS.CNF
Informix 7	SQLINF32.DLL and SQL_INF.CNF
Informix 9	SQLINF9.DLL and SQL_INF9.CNF
DB/2	SQLDB232.DLL and SQL_DB2.CNF
InterBase	SQLINT32.DLL and SQL_INT.CNF

Install SQL Links using InstallShield Express or other certified installation program. For specific information concerning the installation and configuration of SQL Links, see the help file SQLLNK32.HLP, by default installed into the main BDE directory.

Deploying multi-tiered database applications (DataSnap)

DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

Install DataSnap along with a multi-tier application using InstallShield Express (or other Borland-certified installation scripting utility). See the DEPLOY document (found in the main Delphi directory) for details on the files that need to be redistributed with an application. Also see the REMOTE document for related information on what DataSnap files can be redistributed and how.

Deploying Web applications

Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension DLLs (ISAPI and Apache), CGI applications, and ActiveForms.

The steps for deploying Web applications are the same as those for general applications, except the application's files are deployed on the Web server. For information on installing general applications, see "Deploying general applications" on page 13-1. For information on deploying database Web applications, see "Deploying database applications" on page 13-6.

Here are some special considerations for deploying Web applications:

- For BDE database applications, the Borland Database Engine (or alternate database engine) is installed with the application files on the Web server.
- For dbExpress applications, the dbExpress DLLs must be included in the path. If included, the dbExpress driver is installed with the application files on the Web server.
- Security for the directories should be set so that the application can access all needed database files.
- The directory containing an application must have read and execute attributes.
- The application should not use hard-coded paths for accessing database or other files.
- The location of an ActiveX control is indicated by the CODEBASE parameter of the <OBJECT> HTML tag.

Deployment on Apache is described in the next section.

Deployment on Apache

WebBroker supports Apache version 1.3.9 and later for DLLs and CGI applications. Apache is configured by files in the conf directory.

If creating Apache DLLs, you need to be sure to set appropriate directives in the Apache server configuration file, called httpd.conf. The DLL should be physically located in the Modules subdirectory of the Apache software.

If creating CGI applications, the physical directory (specified in the Directory directive of the httpd.conf file) must have the ExecCGI option set to allow execution of programs so the CGI script can be executed. To ensure that permissions are set up properly, you need to either use the ScriptAlias directive or set Options ExecCGI to on.

The ScriptAlias directive creates a virtual directory on your server and marks the target directory as containing CGI scripts. For example, you could add the following line to your httpd.conf file:

```
ScriptAlias /cgi-bin "c:\inetpub\cgi-bin"
```

This would cause requests such as /cgi-bin/mycgi to be satisfied by running the script c:\inetpub\cgi-bin\mycgi.

You can also set Options to All or to ExecCGI using the Directory directive in httpd.conf. The Options directive controls which server features are available in a particular directory. Directory directives are used to enclose a set of directives that apply to the named directory and its subdirectories. An example of the Directory directive is shown below:

```
<Directory <apache-root-dir>\cgi-bin>
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

In this example, Options is set to ExecCGI permitting execution of CGI scripts in the directory cgi-bin.

Note Apache executes locally on the server within the account specified in the User directive in the httpd.conf file. Make sure that the user has the appropriate rights to access the resources needed by the application.

Information concerning the deployment of Apache software can be found in the Apache LICENSE file, which is included in the Apache distribution. You can also find configuration information on the Apache Web site at www.apache.org.

Programming for varying host environments

Due to the nature of various operating system environments, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths
- Fonts
- Operating systems versions
- Helper applications
- DLL locations

Screen resolutions and color depths

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

Considerations when not dynamically resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution.

Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.

Considerations when dynamically resizing forms and controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

- The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen.Height* or the *TScreen.Width* property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.
- Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *CustomForm.Scaled* property to *True* and calling the *TWinControl.ScaleBy* method (*TWidgetControl.ScaleBy* for cross-platform applications). The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.
- The controls on a form can be resized manually, instead of automatically with the *TWinControl.ScaleBy* method (*TWidgetControl.ScaleBy* for cross-platform applications), by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.
- If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 (640 / 1024 = 0.625). The original font size of 8 is reduced to 5 (8 * 0.625 = 5). Text in the application appears jagged and unreadable as it is displayed in the reduced font size.

- Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen resolutions. This effect is offset by setting the *AutoSize* property of these controls to *False*.
- Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

Accommodating varying color depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

Fonts

The Windows and Linux operating systems come with a standard sets of fonts. When designing an application to be deployed on other computers, realize that not all computers will have fonts outside the standard sets.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Windows has a safety measure to account for attempts to use a font that does not exist on the computer. It substitutes another, existing font that it considers the closest match. While this may circumvent errors concerning missing fonts, the end result may be a degradation of the visual appearance of the application. It is better to prepare for this eventuality at design time.

To make a nonstandard font available to a Windows application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .fot file for the nonstandard font with the application.

Operating systems versions

When using operating system APIs or accessing areas of the operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different operating system versions.

To account for this possibility, you have a few options:

- Specify in the application's system requirements the versions of the operating system on which the application can run. It is the user's responsibility to install and use the application only under compatible operating system versions.
- Check the version of the operating system as the application is installed. If an incompatible version of the operating system is present, either halt the installation process or at least warn the installer of the problem.
- Check the operating system version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of the operating system is present, abort the process and alert the user. Alternately, provide different code to run dependent on different operating system versions. For example, some operations are performed differently on Windows 95/98 than on Windows NT/2000. Use the Windows API function *GetVersionEx* to determine the Windows version.

Software license requirements

The distribution of some files associated with Delphi applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:

- DEPLOY
- README
- No-nonsense license agreement
- Third-party product documentation

DEPLOY

DEPLOY covers the some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with a Delphi application. DEPLOY is a document installed in the main Delphi directory. The topics covered include

- .exe, .dll, and .bpl files
- Components and design-time packages
- Borland Database Engine (BDE)
- ActiveX controls
- Sample Images
- SQL Links

README

README contains last minute information about Delphi, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. README is a document installed into the main Delphi directory.

No-nonsense license agreement

The Delphi no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Delphi.

Third-party product documentation

Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Delphi applications prior to distribution.

Part

II

Developing database applications

The chapters in “Developing Database Applications” present concepts and skills necessary for creating Delphi database applications.

Note You need the Professional or Enterprise edition of Delphi to develop database applications. To implement more advanced Client/Server databases, you need the Delphi features available in the Enterprise edition.

Chapter

14

Designing database applications

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify existing data.

This chapter introduces some common considerations for designing a database application and the decisions involved in designing a user interface.

Using databases

Delphi includes many components for accessing databases and representing the information they contain. They are grouped according to the data access mechanism:

- The BDE page of the component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the BDE components, see Chapter 20, “Using the Borland Database Engine.”
- The ADO page of the component palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. There is a broad range of ADO drivers available for connecting to different database servers. Using ADO-based components lets you

integrate your application into an ADO-based environment (for example, making use of ADO-based application servers). For more information about using the ADO components, see Chapter 21, “Working with ADO components”.

- The dbExpress page of the component palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. In addition, dbExpress components support cross-platform development because they are also available on Linux. However, dbExpress database components also support the narrowest range of data manipulation functions. For more information about using the dbExpress components, see Chapter 22, “Using unidirectional datasets”.
- The InterBase page of the Component palette contains components that access InterBase databases directly, without going through a separate engine layer.
- The Data Access page of the component palette contains components that can be used with any data access mechanism. This page includes *TClientDataset*, which can work with data stored on disk or, using the *TDataSetProvider* component also on this page, with components from one of the other groups. For more information about using client datasets, see Chapter 23, “Using client datasets”. For more information about *TDataSetProvider*, see Chapter 24, “Using provider components”.

Note Different versions of Delphi include different drivers for accessing database servers using the BDE, ADO, or dbExpress.

When designing a database application, you must decide which set of components to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

In addition to choosing a data access mechanism, you must choose a database server. There are different types of databases and you will want to consider the advantages and disadvantages of each type before settling on a particular database server.

All types of databases contain tables which store information. In addition, most (but not all) servers support additional features such as

- Database security
- Transactions
- Referential integrity, stored procedures, and triggers

Types of databases

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Delphi provides support for two types of relational database server:

- **Remote database servers** reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote

Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique “dialect” of SQL. Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered applications** because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions. Some local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.
- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.
- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are less expensive to operate because they do not require separately installed servers or expensive site licenses.

Database security

Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBASE, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.

Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see “Controlling server login” on page 17-4.

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can’t be breached by reading the password from the application.

If you require your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password at the point when you will eventually log in to the SQL database, rather than when opening individual tables.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password that is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use HTTPs, CORBA, or COM+ to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover the durable state after disk media failures. Transactions also form the basis of multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user’s commands can’t disrupt the unity of another user’s transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is not part of most local databases, although it is provided by local InterBase. In addition, the BDE drivers provide limited transaction support for some local databases. Database transaction support is provided by the component that represents the database connection. For details on managing transactions using a database connection component, see “Managing transactions” on page 17-5.

In multi-tiered applications, you can create transactions that include actions other than database operations or that span multiple databases. For details on using transactions in multi-tiered applications, see “Managing transactions in multi-tiered applications” on page 25-18.

Referential integrity, stored procedures, and triggers

All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

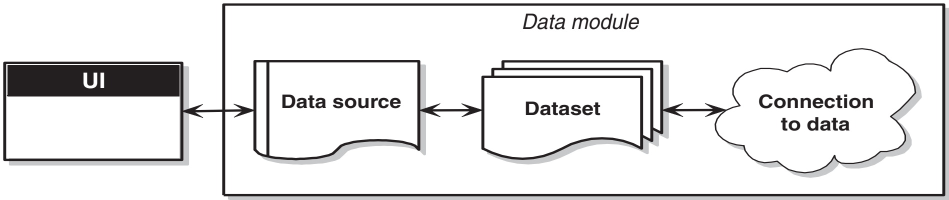
Database architecture

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

General structure

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in Figure 14.1:

Figure 14.1 Generic Database Architecture



The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see “Designing the user interface” on page 14-15.

The data module

If you have isolated your user interface into its own form, you can use a data module to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

The data source

The first item in the data module is a data source. The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Chapter 18, “Understanding datasets”.

The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. There are four basic mechanisms for connecting to the data:

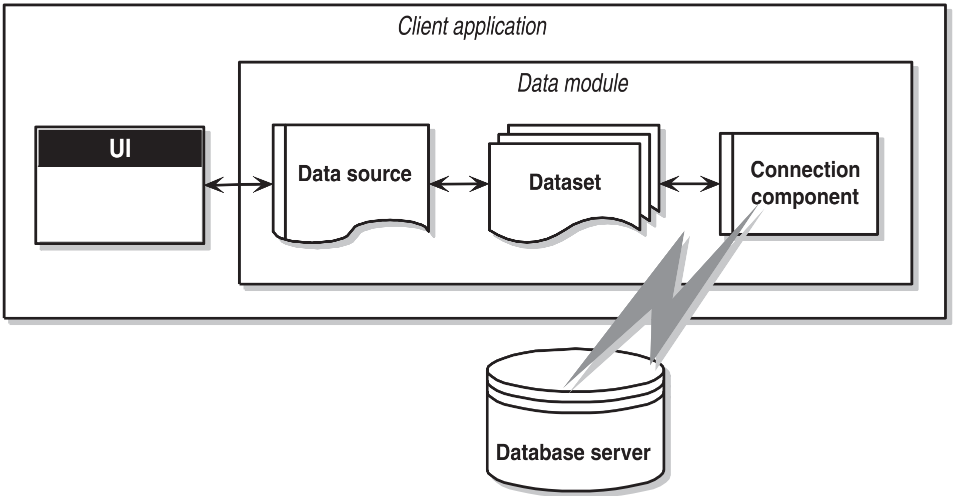
- Connecting directly to a database server. Most datasets use a descendant of *TCustomConnection* to represent the connection to a database server.
- Using a dedicated file on disk. Client datasets support the ability to work with a dedicated file on disk. No separate connection component is needed when working with a dedicated file because the client dataset itself knows how to read from and write to the file.
- Connecting to another dataset. Client datasets can work with data provided by another dataset. A *TDataSetProvider* component serves as an intermediary between the client dataset and its source dataset. This dataset provider can reside in the same data module as the client dataset, or it can be part of an application server running on another machine. If the provider is part of an application server, you also need a special descendant of *TCustomConnection* to represent the connection to the application server.
- Obtaining data from an RDS DataSpace object. ADO datasets can use a *TRDSConnection* component to marshal data in multi-tier database applications that are built using ADO-based application servers.

Sometimes, these mechanisms can be combined in a single application.

Connecting directly to a database server

The most common database architecture is the one where the dataset uses a connection component to establish a connection to a database server. The dataset then fetches data directly from the server and posts edits directly to the server. This is illustrated in Figure 14.2.

Figure 14.2 Connecting directly to the database server



Each type of dataset uses its own type of connection component, which represents a single data access mechanism:

- If the dataset is a BDE dataset such as *TTable*, *TQuery*, or *TStoredProc*, the connection component is a *TDataBase* object. You connect the dataset to the database component by setting its *Database* property. You do not need to explicitly add a database component when using BDE dataset. If you set the dataset's *DatabaseName* property, a database component is created for you automatically at runtime.
- If the dataset is an ADO dataset such as *TADODataSet*, *TADOTable*, *TADOQuery*, or *TADOStoredProc*, the connection component is a *TADOConnection* object. You connect the dataset to the ADO connection component by setting its *ADOConnection* property. As with BDE datasets, you do not need to explicitly add the connection component: instead you can set the dataset's *ConnectionString* property.
- If the dataset is a dbExpress dataset such as *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, or *TSQLStoredProc*, the connection component is a *TSQLConnection* object. You connect the dataset to the SQL connection component by setting its *SQLConnection* property. When using dbExpress datasets, you must explicitly add the connection component. Another difference between dbExpress datasets and the other datasets is that dbExpress datasets are always read-only and unidirectional: This means you can only navigate by iterating through the records in order, and you can't use the dataset methods that support editing.
- If the dataset is an InterBase Express dataset such as *TIBDataSet*, *TIBTable*, *TIBQuery*, or *TIBStoredProc*, the connection component is a *TIBDatabase* object. You connect the dataset to the IB database component by setting its *Database* property. As with dbExpress datasets, you must explicitly add the connection component.

In addition to the components listed above, you can use a specialized client dataset such as *TBDEClientDataSet*, *TSQLClientDataSet*, or *TIBClientDataSet* with a database connection component. When using one of these client datasets, specify the appropriate type of connection component as the value of the *DBConnection* property.

Although each type of dataset uses a different connection component, they all perform many of the same tasks and surface many of the same properties, methods, and events. For more information on the commonalities among the various database connection components, see Chapter 17, “Connecting to databases”.

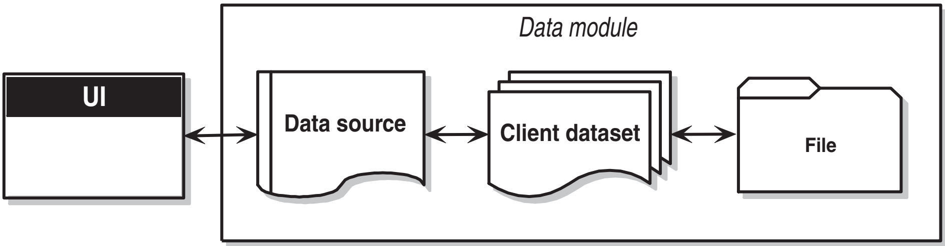
This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

Note The connection components or drivers needed to create two-tiered applications are not available in all version of Delphi.

Using a dedicated file on disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses MyBase, the ability of client datasets to save themselves to a file and to load the data from a file. This architecture is illustrated in Figure 14.3:

Figure 14.3 A file-based database application



When using this file-based approach, your application writes changes to disk using the client dataset’s *SaveToFile* method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the *LoadFromFile* method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the *FileName* property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (If you do not statically link in midaslib.dcu, the client dataset does require midas.dll). There is no need for site licenses or database administration.

In addition, some versions of Delphi let you convert between arbitrary XML documents and the data packets that are used by a client dataset. Thus, the file-based approach can be used to work with XML documents as well as dedicated datasets. For information about converting between XML documents and client dataset data packets, see Chapter 26, “Using XML in database applications”.

The file-based approach offers no support for multiple users. The dataset should be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other’s data files.

For more information about using a client dataset with data stored on disk, see “Using a client dataset with file-based data” on page 23-31.

Connecting to another dataset

There are specialized client datasets that use the BDE or *dbExpress* to connect to a database server. These specialized client datasets are, in fact, composite components that include another dataset internally to access the data and an internal provider component to package the data from the source dataset and to apply updates back to the database server. These composite components require some additional overhead, but provide certain benefits:

- Client datasets provide the most robust way to work with cached updates. By default, other types of datasets post edits directly to the database server. You can reduce network traffic by using a dataset that caches updates locally and applies them all later in a single transaction. For information on the advantages of using client datasets to cache updates, see “Using a client dataset to cache updates” on page 23-15.
- Client datasets can apply edits directly to a database server when the dataset is read-only. When using *dbExpress*, this is the only way to edit the data in the dataset (it is also the only way to navigate freely in the data when using *dbExpress*). Even when not using *dbExpress*, the results of some queries and all stored procedures are read-only. Using a client dataset provides a standard way to make such data editable.
- Because client datasets can work directly with dedicated files on disk, using a client dataset can be combined with a file-based model to allow for a flexible “briefcase” application. For information on the briefcase model, see “Combining approaches” on page 14-14.

In addition to these specialized client datasets, there is a generic client dataset (*TClientDataSet*), which does not include an internal dataset and dataset provider. Although *TClientDataSet* has no built-in database access mechanism, you can connect it to another, external, dataset from which it fetches data and to which it sends updates. Although this approach is a bit more complicated, there are times when it is preferable:

- Because the source dataset and dataset provider are external, you have more control over how they fetch data and apply updates. For example, the provider component surfaces a number of events that are not available when using a specialized client dataset to access data.
- When the source dataset is external, you can link it in a master/detail relationship with another dataset. An external provider automatically converts this arrangement into a single dataset with nested details. When the source dataset is internal, you can't create nested detail sets this way.
- Connecting a client dataset to an external dataset is an architecture that easily scales up to multiple tiers. Because the development process can get more involved and expensive as the number of tiers increases, you may want to start developing your application as a single-tiered or two-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. If you think you may eventually use a multi-tiered architecture, it can be worthwhile to start by using a client dataset with an external source dataset. This way, when you move the data access and manipulation logic to a middle tier, you protect your development investment because the code can be reused as your application grows.
- *TClientDataSet* can link to any source dataset. This means you can use custom datasets (third-party components) for which there is no corresponding specialized client dataset. Some versions of Delphi even include special provider components that connect a client dataset to an XML document rather than another dataset. (This works the same way as connecting a client dataset to another (source) dataset, except that the XML provider uses an XML document rather than a dataset. For information about these XML providers, see "Using an XML document as the source for a provider" on page 26-8.)

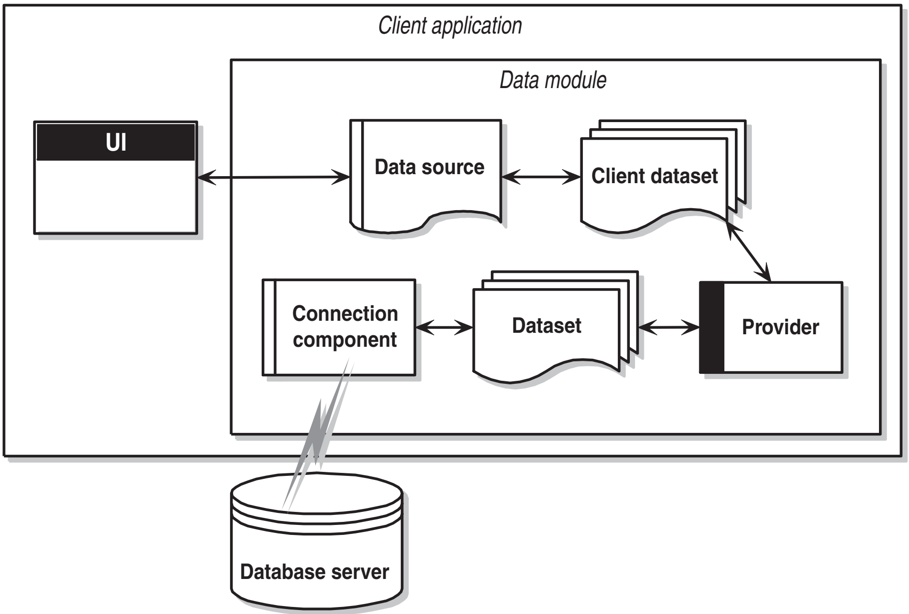
There are two versions of the architecture that connects a client dataset to an external dataset:

- Connecting a client dataset to another dataset in the same application.
- Using a multi-tiered architecture.

Connecting a client dataset to another dataset in the same application

By using a provider component, you can connect *TClientDataSet* to another (source) dataset. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. The architecture for this is illustrated in Figure 14.4:

Figure 14.4 Architecture combining a client dataset and another dataset



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

To link the client dataset to the provider, set its *ProviderName* property to the name of the provider component. The provider must be in the same data module as the client dataset. To link the provider to the source dataset, set its *DataSet* property.

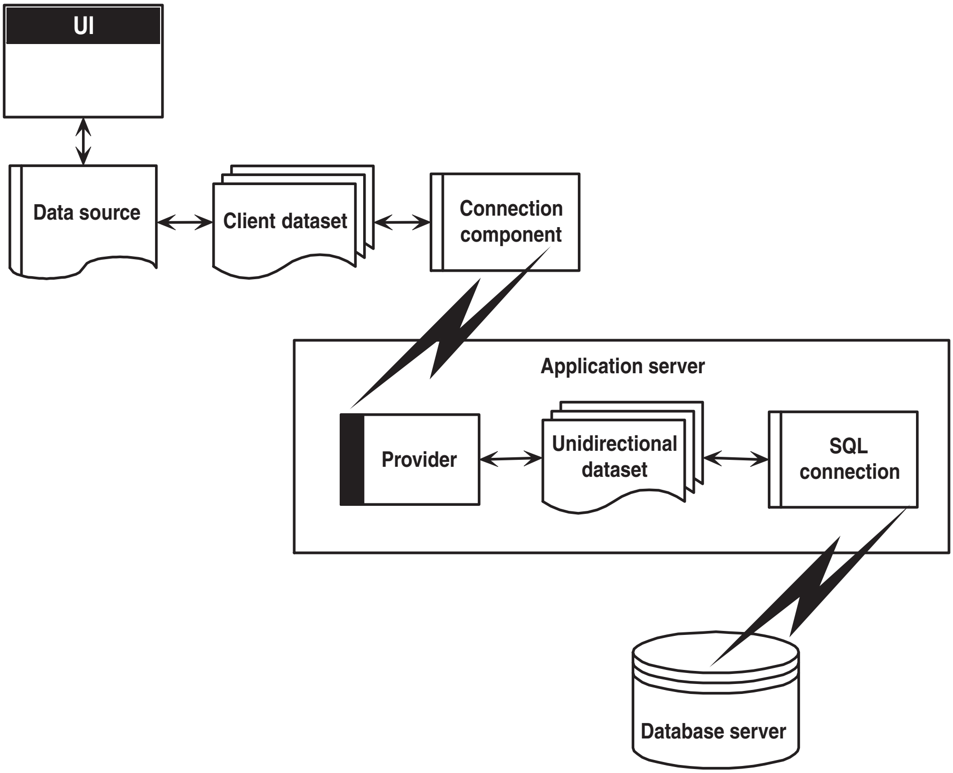
Once the client dataset is linked to the provider and the provider is linked to the source dataset, these components automatically handle all the details necessary for fetching, displaying, and navigating through the database records (assuming the source dataset is connected to a database). To apply user edits back to the database, you need only call the client dataset's *ApplyUpdates* method.

For more information on using a client dataset with a provider, see "Using a client dataset with a provider" on page 23-23.

Using a multi-tiered architecture

When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a multi-tiered application. Multi-tiered applications have middle tiers between the client application and database server. The architecture for this is illustrated in Figure 14.5:

Figure 14.5 Multi-tiered database architecture



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data, while ensuring consistent data logic. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading data-processing over several systems.

The multi-tiered architecture is very similar to the previous model. It differs mainly in that source dataset that connects to the database server and the provider that acts as an intermediary between that source dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the "remote data broker").

Because the provider has moved to a separate application, the client dataset can no longer connect to the source dataset by simply setting its *ProviderName* property. In addition, it must use some type of connection component to locate and connect to the application server.

There are several types of connection components that can connect a client dataset to an application server. They are all descendants of *TCustomRemoteServer*, and differ primarily in the communication protocol they use (TCP/IP, HTTP, DCOM, SOAP, or CORBA). Link the client dataset to its connection component by setting the *RemoteServer* property.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

For more information about connecting a client dataset to an application server, see Chapter 25, "Creating multi-tiered applications".

Combining approaches

The previous sections describe several architectures you can use when writing database applications. There is no reason, however, why you can't combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in "Using a dedicated file on disk" on page 14-9 with another approach such as those described in "Connecting a client dataset to another dataset in the same application" on page 14-11 or "Using a multi-tiered architecture" on page 14-12. These combinations are easy because all models use a client dataset to represent the data that appears in the user interface. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An onsite company database contains customer contact data that sales representatives can use and update in the field. While onsite, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales reps return onsite, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The client dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back onsite, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

Designing the user interface

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see Chapter 15, "Using data controls."

In addition to the basic data controls, you may also want to introduce other elements into your user interface:

- You may want your application to analyze the data contained in a database. Applications that analyze data do more than just display the data in a database, they also summarize the information in useful formats to help users grasp the impact of that data.
- You may want to print reports that provide a hard copy of the information displayed in your user interface.
- You may want to create a user interface that can be viewed from Web browsers. The simplest Web-based database applications are described in "Using database information in responses" on page 28-17. In addition, you can combine the Web-based approach with the multi-tiered architecture, as described in "Writing Web-based client applications."

Analyzing data

Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data.

The *TDBChart* component on the Data Controls page of the Component palette lets you present database information in a graphical format that enables users to quickly grasp the import of database information.

In addition, some versions of Delphi include a Decision Cube page on the Component palette. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information about using the Decision Cube components, see Chapter 16, "Using decision support components".

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates with a client dataset. For more information about using maintained aggregates, see "Using maintained aggregates" on page 23-11.

Writing reports

If you want to let your users print database information from the datasets in your application, you can use the report components on the QReport page of the Component palette. Using these components you can visually build banded reports to present and summarize the information in your database tables. You can add summaries to group headers or footers to analyze the data based on grouping criteria.

Start a report for your application by selecting the QuickReport icon from the New Items dialog. Select File|New from the main menu, and go to the page labeled Business. Double-click the QuickReport Wizard icon to launch the wizard.

Note See the QuickReport demo that ships with Delphi for an example of how to use the components on the QReport page.

Chapter

15

Using data controls

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in "Displaying a single record" on page 15-7.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. "Choosing how to organize the data" on page 15-7 describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator. For more information about using a data navigator, see "Navigating and manipulating records" on page 15-28.

Note More complex data-aware controls for decision support are discussed in Chapter 16, "Using decision support components."

Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described below.

Using common data control features

The following tasks are common to most data controls:

- Associating a data control with a dataset
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display
- Enabling mouse, keyboard, and timer events

Data controls let you display and edit fields of data associated with the current record in a dataset. Table 15.1 summarizes the data controls that appear on the Data Controls page of the Component palette.

Table 15.1 Data controls

Data control	Description
TDBGrid	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
TDBNavigator	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
TDBText	Displays data from a field as a label.
TDBEdit	Displays data from a field in an edit box.
TDBMemo	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
TDBImage	Displays graphics from a data field in a graphics box.
TDBListBox	Displays a list of items from which to update a field in the current data record.
TDBComboBox	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
TDBCheckBox	Displays a check box that indicates the value of a Boolean field.
TDBRadioGroup	Displays a set of mutually exclusive options for a field.
TDBLookupListBox	Displays a list of items looked up from another dataset based on the value of a field.
TDBLookupComboBox	Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box.
TDBCtrlGrid	Displays a configurable, repeating set of data-aware controls within a grid.
TDBRichEdit	Displays formatted data from a field in an edit box.

Data controls are data-aware at design time. When you associate the data control with an active dataset while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see "Creating persistent fields" on page 19-4.

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

Associating a data control with a dataset

Data controls connect to datasets by using a data source. A data source component (*TDataSource*) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

Note Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset,

1. Place a dataset in a data module (or on a form), and set its properties as appropriate.
2. Place a data source in the same data module (or form). Using the Object Inspector, set its *DataSet* property to the dataset you placed in step 1.
3. Place a data control from the Data Access page of the Component palette onto a form.
4. Using the Object Inspector, set the *DataSource* property of the control to the data source component you placed in step 2.
5. Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid, TDBCtrlGrid,* and *TDBNavigator* because they access all available fields in the dataset.
6. Set the *Active* property of the dataset to *True* to display data in the control.

Changing the associated dataset at runtime

In the preceding example, the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
    else
      DataSet := Customers;
  end;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
```

Enabling and disabling the data source

The data source has an *Enabled* property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's *DisableControls* and *EnableControls* methods because they affect all attached data sources.

Responding to changes mediated by the data source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The *OnDataChange* event occurs whenever the data in a record may have changed, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The *OnUpdateData* event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the underlying database server or local cache.

The *OnStateChange* event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's *State* property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  ⋮
end;
```

Note For more information about dataset states, see "Determining dataset states" on page 18-3.

Editing and updating data

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

Note Unidirectional datasets never permit users to edit and update data.

Enabling editing in controls on user entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source's *AutoEdit* property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a *TDBNavigator* control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see "Navigating and manipulating records" on page 15-28. Alternately, you can write code that calls the dataset's *Edit* method when you want to put the dataset into edit mode.

Editing data in a control

A data control can only post edits to its associated dataset if the dataset's *CanModify* property is *True*. *CanModify* is always *False* for unidirectional datasets. Some datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*.

Note Whether a dataset can update data depends on whether the underlying database table permits updates.

Even if the dataset's *CanModify* property is *True*, the *Enabled* property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is ***True*** (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property of the data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. Clearly, you will want to ensure that the control's *ReadOnly* property is *True* when the dataset's *CanModify* property is *False*. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying dataset when you Tab from the control. If you press Esc before you Tab from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In *TDBGrid*, modifications are posted when you move to a different record; you can press *Esc* in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware controls associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

Note If your application caches updates (for example, using a client dataset), all modifications are posted to an internal cache. These modifications are not applied to the underlaying database table until you call the dataset's *ApplyUpdates* method.

Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...finally** statement so that you can re-enable controls even if an exception occurs during processing. The **finally** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

Refreshing data display

The *Refresh* method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. If you are using cached updates, before you refresh the dataset you must apply any updates the dataset has currently cached.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

Enabling mouse, keyboard, and timer events

The *Enabled* property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Choosing how to organize the data

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records.

In addition, you will want to add controls to navigate and manipulate records. The *TDBNavigator* control provides built-in support for many of the functions you may want to perform.

Displaying a single record

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. These controls are typically data-aware versions of other controls that are available on the component palette. For example, the *TDBEdit* control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field.

Displaying data as labels

TDBText is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

TDBText gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

Note When you place a *TDBText* component on a form, make sure its *AutoSize* property is *True* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is *False*, and the control is too small, data display is clipped.

Displaying and editing fields in an edit box

TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open *TClientDataSet* called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

Displaying and editing text in a memo control

TDBMemo is a data-aware component—similar to the standard *TMemo* component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the memo control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the *ScrollBars* property. To prevent word wrap, set the *WordWrap* property to ***False***. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the *Font* property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to ***False***, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing text in a rich edit memo control

TDBRichEdit is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBRichEdit* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well.

Note While *TDBRichEdit* provides properties and methods to enter and work with rich text, it does not provide any user interface components to make these formatting options available to the user. Your application must implement the user interface to surface rich text capabilities.

By default, *TDBRichEdit* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the rich edit control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for MaxLength is 0, meaning that there is no character limit other than that imposed by the operating system.

Because the *TDBRichEdit* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBRichEdit* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBRichEdit* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing graphics fields in an image control

TDBImage is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch* property to *True* to resize the graphic to fit within an image control as it is resized.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay* to *False*, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing data in list and combo boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list and combo box controls:

- *TDBListBox*, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- *TDBComboBox*, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- *TDBLookupListBox*, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- *TDBLookupComboBox*, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

Note At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. *Backspace* and *Esc* cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items* property in the Object Inspector. Then type in the items that you want to have appear in the list. At runtime, use the methods of the *Items* property to manipulate its string list.

When a *TDBListBox* or *TDBComboBox* control is linked to a field through its *DataField* property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, *TDBComboBox* displays the current value for the field in its edit box, regardless of whether it appears in the *Items* list.

For *TDBListBox*, the *Height* property determines how many items are visible in the list box at one time. The *IntegralHeight* property controls how the last item can appear. If *IntegralHeight* is ***False*** (the default), the bottom of the list box is determined by the *ItemHeight* property, and the bottom item may not be completely displayed. If *IntegralHeight* is *True*, the visible bottom item in the list box is fully displayed.

For *TDBComboBox*, the *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the *Items* list is displayed at runtime:

- *Style* determines the display style of the component:
  - *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
  - *csSimple*: Combines an edit control with a fixed size list of items that is always displayed. When setting *Style* to *csSimple*, be sure to increase the *Height* property so that the list is displayed.
  - *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
  - *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.
- *DropDownCount*: the maximum number of items displayed in the list. If the number of *Items* is greater than *DropDownCount*, the user can scroll the list. If the number of *Items* is less than *DropDownCount*, the list will be just large enough to display all the Items.
- *ItemHeight*: The height of each item when style is *csOwnerDrawFixed*.
- *Sorted*: If *True*, then the *Items* list is displayed in alphabetical order.

Displaying and editing data in lookup list and combo boxes

Lookup list boxes and lookup combo boxes (*TDBLookupListBox* and *TDBLookupComboBox*) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

- **A lookup field defined for a dataset.**
  To specify list box items using a lookup field, the dataset to which you link the

control must already define a lookup field. (This process is described in “Defining a lookup field” on page 19-8). To specify the lookup field for the list box items,

1. Set the *DataSource* property of the list box to the data source for the dataset containing the lookup field to use.
2. Choose the lookup field to use from the drop-down list for the *DataField* property.

When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

- **A secondary data source, data field, and key**.
If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item. To specify a secondary data source for list box items,

1. Set the *DataSource* property of the list box to the data source for the control.
2. Choose a field into which to insert looked-up values from the drop-down list for the *DataField* property. The field you choose cannot be a lookup field.
3. Set the *ListSource* property of the list box to the data source for the dataset that contain the field whose values you want to look up.
4. Choose a field to use as a lookup key from the drop-down list for the *KeyField* property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
5. Choose a field whose values to return from the drop-down list for the *ListField* property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the *RowCount* property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the *DropDownRows* property instead.

Note You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see “Defining a lookup list column” on page 15-20.

Handling Boolean field values with check boxes

TDBCheckBox is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of *ValueChecked* and *ValueUnchecked* properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

Note The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to “true,” but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the field for the current record contains values of “true,” “Yes,” or “On,” then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to “false,” but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

Restricting field values with radio controls

TDBRadioGroup is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items* property determines the radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button’s label.

If the current value of a field associated with a radio group matches one of the strings in the *Items* property, that radio button is selected. For example, if three strings, “Red,” “Yellow,” and “Blue,” are listed for *Items*, and the field for the current record contains the value “Blue,” then the third button in the group appears selected.

Note If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the *Values* property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains "Red," "Yellow," and "Blue," and *Values* contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

If strings for *Values* are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

Displaying multiple records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in "Viewing and editing data with TDBGrid" on page 15-15 and "Creating a grid that contains other data-aware controls" on page 15-26.

Note You can't display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

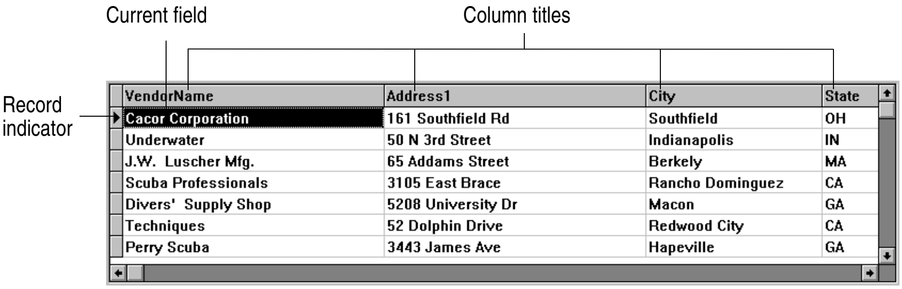
- **Master-detail forms:** You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see "Creating master/detail relationships" on page 18-34 and "Establishing master/detail relationships using parameters" on page 18-46.
- **Drill-down forms**: In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

Tip It is generally not a good idea to combine these two approaches on a single form. It is usually confusing for users to understand the data relationships in such forms.

Viewing and editing data with TDBGrid

A *TDBGrid* control lets you view and edit records in a dataset in a tabular grid format.

Figure 15.1 TDBGrid control



VendorName	Address1	City	State
Cacor Corporation	161 Southfield Rd	Southfield	OH
Underwater	50 N 3rd Street	Indianapolis	IN
J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
Divers' Supply Shop	5208 University Dr	Macon	GA
Techniques	52 Dolphin Drive	Redwood City	CA
Perry Scuba	3443 James Ave	Hapeville	GA

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see "Creating a customized grid" on page 15-16.
- Creation of persistent field components for the dataset displayed in the grid. For more information about creating persistent field components using the Fields editor, see Chapter 19, "Working with field components."
- The dataset's *ObjectView* property setting for grids displaying ADT and array fields. See "Displaying ADT and array fields" on page 15-21.

A grid control has a *Columns* property that is itself a wrapper on a *TDBGridColumns* object. *TDBGridColumns* is a collection of *TColumn* objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the *Columns* property of the grid to access the properties, events, and methods of *TDBGridColumns* at runtime.

Using a grid control in its default state

The *State* property of the grid's *Columns* property indicates whether persistent column objects exist for the grid. *Columns.State* is a runtime-only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.

When the grid's *Columns.State* property is *csDefault*, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Paradox table at one moment, then switch to display the results of an SQL query when the grid's *DataSource* property changes or when the *DataSet* property of the data source itself is changed.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the *Width* property of a column changes the *DisplayWidth* property of the field associated with that column. Changes made to column properties that are not based on field properties, such as *Font*, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

Note Changing a grid's *Columns.State* property to *csDefault* at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.

Creating a customized grid

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property. If you modify the *DisplayLabel* property, the column title reflects that change immediately. If you then assign a string to the column title's caption, the tile caption becomes independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of the record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

Note Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's *FieldCount* property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's *SelectedField* property is NULL and the *SelectedIndex* property is –1.

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (…) in a cell that can be clicked upon to launch special data viewers or dialogs related to the current cell.

Creating persistent columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the *State* property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control,

1. Select the grid component in the form.
2. Invoke the Columns editor by double clicking on the grid's *Columns* property in the Object Inspector.

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis. To create persistent columns for all fields:

1. Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.
2. If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.
3. Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually:

1. Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).
2. To associate a field with this new column, set the *FieldName* property in the Object Inspector.
3. To set the title for the new column, expand the *Title* property in the Object Inspector and set its *Caption* property.
4. Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

Deleting persistent columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display. To remove a persistent column from a grid,

1. Double-click the grid to display the Columns editor.
2. Select the field to remove in the Columns list box.
3. Click Delete (you can also use the context menu or *Del* key, to remove a column).

Note If you delete all the columns from a grid, the *Columns.State* property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
```

Arranging the order of persistent columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column,

1. Select the column in the Columns list box.
2. Drag it to a new location in the list box.

You can also change the column order by clicking on the column title of the actual grid and dragging the column to a new position, just as you can at runtime.

Note Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

Important You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's *OnColumnMoved* event fires after a column has been moved.

Setting column properties at design time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component (called the *default source*) such as a grid or an associated field component.

To set a column's properties, select the column in The Columns editor and set its properties in the Object Inspector. The following table summarizes key column properties you can set.

Table 15.2 Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: *TField.Alignment*.
ButtonStyle	*cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data. *cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event. *cbsNone*: The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. Default source: *TDBGrid.Color*. (For text foreground color, see the Font property.)

Table 15.2 Column properties (continued)

Property	Purpose
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.
Expanded	Specifies whether the column is expanded. Only applies to columns representing ADT or array fields.
FieldName	Specifies the field name associated with this column. This can be blank.
ReadOnly	*True*: The data in the column cannot be edited by the user. *False*: (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default source: *TField.DisplayWidth*.
Font	Specifies the font type, size, and color used to draw text in the column. Default source: *TDBGrid.Font*.
PickList	Contains a list of values to display in a drop-down list in the column.
Title	Sets properties for the title of the selected column.

The following table summarizes the options you can specify for the *Title* property.

Table 15.3 Expanded TColumn Title properties

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default source: *TField.DisplayLabel*.
Color	Specifies the background color used to draw the column title cell. Default source: *TDBGrid.FixedColor*.
Font	Specifies the font type, size, and color used to draw text in the column title. Default source: *TDBGrid.TitleFont*.

Defining a lookup list column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control. To specify that the column acts like a combo box, set the column's *ButtonStyle* property to *cbsAuto*. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field in the dataset. For information about creating lookup fields, see "Defining a lookup field" on page 19-8. Once the lookup field is defined, set the column's *FieldName* to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the *PickList* property for the column in the Object Inspector. This brings up the String List editor, where you can enter the values that populate the pick list for the column.

By default, the drop-down list displays 7 values. You can change the length of this list by setting the *DropDownRows* property.

Note To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

Putting a button in a column

A column can display an ellipsis button (…) to the right of the normal cell editor. *Ctrl+Enter* or a mouse click fires the grid's *OnEditButtonClick* event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column:

1 Select the column in the *Columns* list box.

2 Set *ButtonStyle* to *cbsEllipsis*.

3 Write an *OnEditButtonClick* event handler.

Restoring default values to a column

At runtime you can test a column's *AssignedValues* property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's *RestoreDefaults* method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1.Columns.RestoreDefaults;
```

Displaying ADT and array fields

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" the field so that each of the simpler types that make up the field appears as a separate field in the dataset. When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

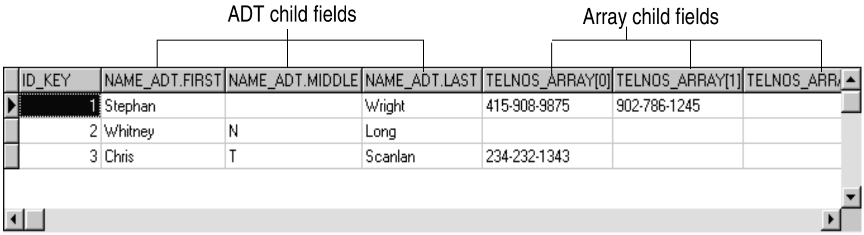
To display composite fields as if they were flattened out, set the dataset's *ObjectView* property to *False*. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.

- It can display composite fields in a single column, reflecting the fact that they are a single field. When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expanded* property of the column:
    - When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
    - When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

  To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to *True*. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse

Figure 15.2 shows a grid with an ADT field and an array field. The dataset's *ObjectView* property is set to *False* so that each child field has a column.

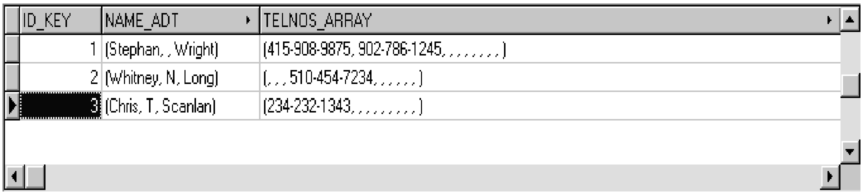
Figure 15.2 TDBGrid control with ObjectView set to False



ID_KEY	NAME_ADT.FIRST	NAME_ADT.MIDDLE	NAME_ADT.LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARR
1	Stephan		Wright	415-908-9875	902-786-1245	
2	Whitney	N	Long			
3	Chris	T	Scanlan	234-232-1343		

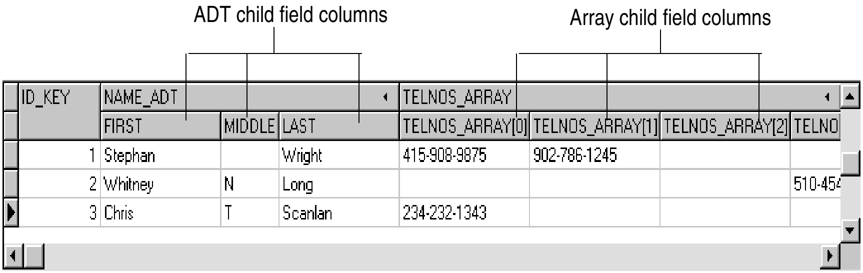
Figure 15.3 and 15.4 show the grid with an ADT field and an array field. Figure 15.3 shows the fields collapsed. In this state they cannot be edited. Figure 15.4 shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.

Figure 15.3 TDBGrid control with Expanded set to False



ID_KEY	NAME_ADT	TELNOS_ARRAY
1	(Stephan, , Wright)	(415-908-9875, 902-786-1245, , , , , , , ,)
2	(Whitney, N, Long)	(, , , 510-454-7234, , , , , ,)
3	(Chris, T, Scanlan)	(234-232-1343, , , , , , , , ,)

Figure 15.4 TDBGrid control with Expanded set to True



ID_KEY	NAME_ADT			TELNOS_ARRAY			
	FIRST	MIDDLE	LAST	TELNOS_ARRAY[0]	TELNOS_ARRAY[1]	TELNOS_ARRAY[2]	TELNO
1	Stephan		Wright	415-908-9875	902-786-1245		
2	Whitney	N	Long				510-454
3	Chris	T	Scanlan	234-232-1343			

The following table lists the properties that affect the way ADT and array fields appear in a *TDBGrid*:

Table 15.4 Properties that affect the way composite fields appear

Property	Object	Purpose
Expandable	TColumn	Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only)
Expanded	TColumn	Specifies whether the column is expanded.
MaxTitleRows	TDBGrid	Specifies the maximum number of title rows that can appear in the grid
ObjectView	TDataSet	Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed.
ParentColumn	TColumn	Refers to the TColumn object that owns the child field's column.

Note In addition to ADT and array fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as “(DataSet)” or “(Reference)”, respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field’s value. For reference fields, this grid contains a single row that displays the record from another dataset.

Setting grid options

You can use the grid *Options* property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of

Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

Table 15.5 Expanded TDBGrid Options properties

Option	Purpose
dgEditing	*True*: (Default). Enables editing, inserting, and deleting records in the grid. *False*: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	*True*: When a field is selected, it is in Edit state. *False*: (Default). A field is not automatically in Edit state when selected.
dgTitles	*True*: (Default). Displays field names across the top of the grid. *False*: Field name display is turned off.
dgIndicator	*True*: (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam. *False*: The indicator column is turned off.
dgColumnResize	*True*: (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying *TField* component. *False*: Columns cannot be resized in the grid.
dgColLines	*True*: (Default). Displays vertical dividing lines between columns. *False*: Does not display dividing lines between columns.
dgRowLines	*True*: (Default). Displays horizontal dividing lines between records. *False*: Does not display dividing lines between records.
dgTabs	*True*: (Default). Enables tabbing between fields in records. *False*: Tabbing exits the grid control.
dgRowSelect	*True*: The selection bar spans the entire width of the grid. *False*: (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	*True*: (Default). The selection bar in the grid is always visible, even if another control has focus. *False*: The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	*True*: (Default). Prompt for confirmation to delete records (*Ctrl+Del*). *False*: Delete records without confirmation.
dgCancelOnExit	*True*: (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records. *False*: Permits pending inserts.
dgMultiSelect	*True*: Allows user to select noncontiguous rows in the grid using *Ctrl+Shift* or *Shift+ arrow* keys. *False*: (Default). Does not allow user to multi-select rows.

Editing in the grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The *CanModify* property of the *Dataset* is *True*.
- The *ReadOnly* property of grid is *False*.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus is changed to another control on a form, the grid does not post changes until another the cursor for the dataset is moved to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, the grid raises an exception, and does not modify the record.

Note If your application caches updates, posting record changes only adds them to an internal cache. They are not posted back to the underlying database table until your application applies the updates.

You can cancel all edits for a record by pressing *Esc* in any field before moving to another record.

Controlling grid drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat* properties of the field component associated with the column.

You can augment the default grid display logic with code in a grid's *OnDrawColumnCell* event. If the grid's *DefaultDrawing* property is *True*, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special graphics in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to *False* and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* inside your *OnDrawColumnCell* event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

Responding to user actions at runtime

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the Object Inspector.

Table 15.6 Grid control events

Event	Purpose
OnCellClick	Occurs when a user clicks on a cell in the grid.
OnColEnter	Occurs when a user moves into a column on the grid.
OnColExit	Occurs when a user leaves a column on the grid.
OnColumnMoved	Occurs when the user moves a column to a new location.
OnDblClick	Occurs when a user double clicks in the grid.
OnDragDrop	Occurs when a user drags and drops in the grid.
OnDragOver	Occurs when a user drags over the grid.
OnDrawColumnCell	Occurs when application needs to draw individual cells.
OnDrawDataCell	(obsolete) Occurs when application needs to draw individual cells if *State* is *csDefault*.
OnEditButtonClick	Occurs when the user clicks on an ellipsis button in a column.
OnEndDrag	Occurs when a user stops dragging on the grid.
OnEnter	Occurs when the grid gets focus.
OnExit	Occurs when the grid loses focus.
OnKeyDown	Occurs when a user presses any key or key combination on the keyboard when in the grid.
OnKeyPress	Occurs when a user presses a single alphanumeric key on the keyboard when in the grid.
OnKeyUp	Occurs when a user releases a key when in the grid.
OnStartDrag	Occurs when a user starts dragging on the grid.
OnTitleClick	Occurs when a user clicks the title for a column.

There are many uses for these events. For example, you might write a handler for the *OnDblClick* event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the *SelectedField* property to determine to current row and column.

Creating a grid that contains other data-aware controls

A *TDBCtrlGrid* control displays multiple fields in multiple records in a tabular grid format. Each cell in a grid displays multiple fields from a single row. To use a database control grid:

1. Place a database control grid on a form.
2. Set the grid's *DataSource* property to the name of a data source.

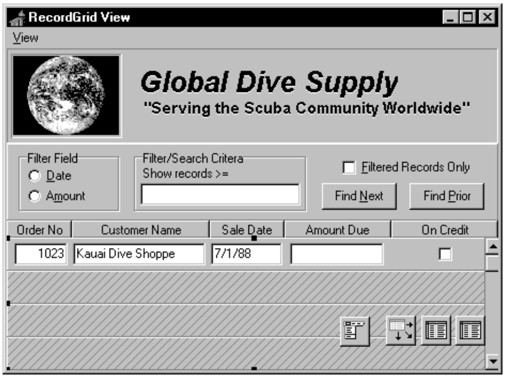
3 Place individual data controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.

4 Set the *DataField* property for each data control to the name of a field. The data source for these data controls is already set to the data source of the database control grid.

5 Arrange the controls within the cell as desired.

When you compile and run an application containing a database control grid, the arrangement of data controls you set in the design cell at runtime is replicated in each cell of the grid. Each cell displays a different record in a dataset.

Figure 15.5 TDBCtrlGrid at design time



The following table summarizes some of the unique properties for database control grids that you can set at design time:

Table 15.7 Selected database control grid properties

Property	Purpose
AllowDelete	*True* (default): Permits record deletion. *False*: Prevents record deletion.
AllowInsert	*True* (default): Permits record insertion. *False*: Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.
Orientation	*goVertical* (default): Display records from top to bottom. *goHorizontal*: Displays records from left to right.
PanelHeight	Sets the height for an individual panel. Default = 72.
PanelWidth	Sets the width for an individual panel. Default = 200.
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	*True* (default): Displays a focus rectangle around the current record's panel at runtime. *False*: Does not display a focus rectangle.

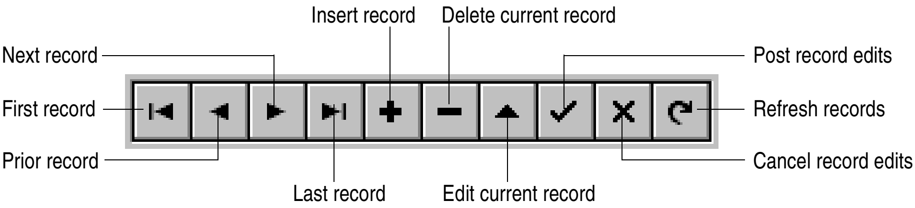
For more information about database control grid properties and methods, see the online *VCL Reference*.

Navigating and manipulating records

TDBNavigator provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

Figure 15.6 shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator enables you to hide or show a subset of these buttons dynamically.

Figure 15.6 Buttons on the TDBNavigator control



The following table describes the buttons on the navigator.

Table 15.8 TDBNavigator buttons

Button	Purpose
First	Calls the dataset's *First* method to set the current record to the first record.
Prior	Calls the dataset's *Prior* method to set the current record to the previous record.
Next	Calls the dataset's *Next* method to set the current record to the next record.
Last	Calls the dataset's *Last* method to set the current record to the last record.
Insert	Calls the dataset's *Insert* method to insert a new record before the current record, and set the dataset in Insert state.
Delete	Deletes the current record. If the *ConfirmDelete* property is *True* it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

Choosing navigator buttons to display

When you first place a *TDBNavigator* on a form at design time, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the

First, *Next*, and *Refresh* buttons are meaningful. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

Hiding and showing navigator buttons at design time

The *VisibleButtons* property in the Object Inspector is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The list of buttons that can be turned on or off appears in the Object Inspector below the *VisibleButtons* property. The + sign changes to a – (minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to *True*, the button appears in the *TDBNavigator*. If ***False***, the button is removed from the navigator at design time and runtime.

Note As button values are set to *False*, they are removed from the *TDBNavigator* on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert, Delete, Edit, Post, Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert, Delete, Edit, Post, Cancel,* and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The *VisibleButtons* property controls which buttons are displayed in the navigator. Here's one way you might code the *OnEnter* event handler:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

Displaying fly-over help

To display fly-over help for each navigator button at runtime, set the navigator *ShowHint* property to *True*. When *ShowHint* is *True*, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default.

The *Hints* property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

Using a single navigator for multiple datasets

As with other data-aware controls, a navigator's *DataSource* property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two edit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the edit control connected to *CustomersSource*, the navigator should also use *CustomersSource*, and when the user enters the edit control connected to *OrdersSource*, the navigator should switch to *OrdersSource* as well. You can code an *OnEnter* event handler for one of the edit controls, and then share that event with the other edit control. For example:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

Chapter

16

Using decision support components

The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see "About crosstabs" on page 16-2.

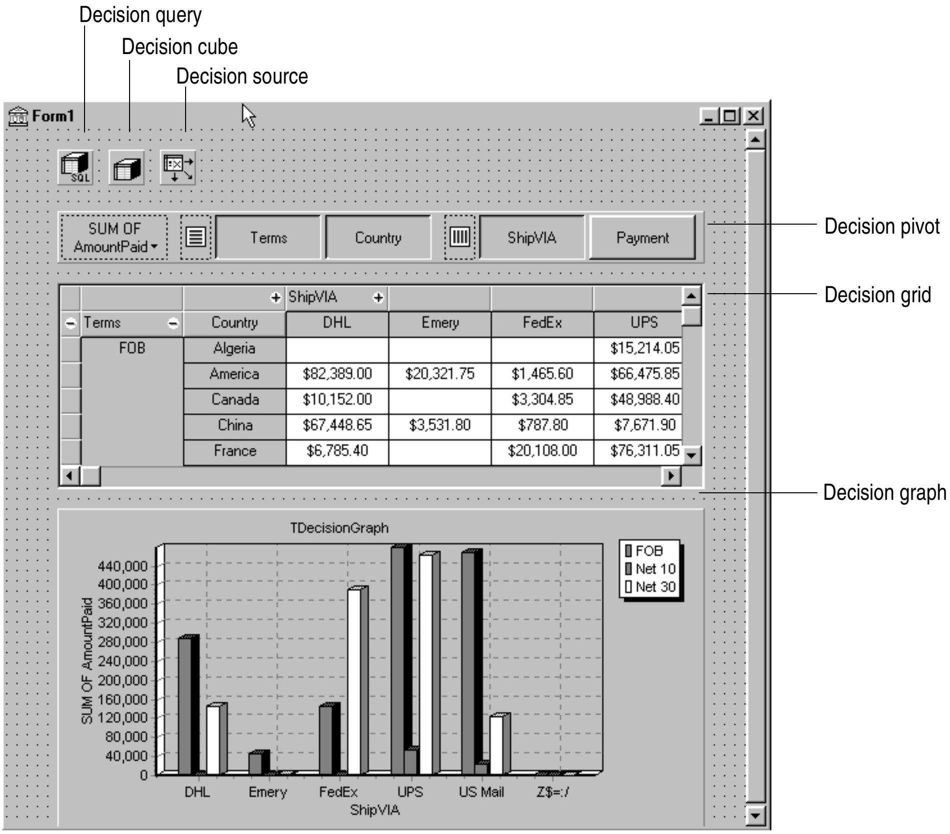
Overview

The decision support components appear on the Decision Cube page of the component palette:

- The decision cube, *TDecisionCube*, is a multidimensional data store.
- The decision source, *TDecisionSource*, defines the current pivot state of a decision grid or a decision graph.
- The decision query, *TDecisionQuery*, is a specialized form of *TQuery* used to define the data in a decision cube.
- The decision pivot, *TDecisionPivot*, lets you open or close decision cube dimensions, or fields, by pressing buttons.
- The decision grid, *TDecisionGrid*, displays single- and multidimensional data in table form.
- The decision graph, *TDecisionGraph*, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified.

Figure 16.1 shows all the decision support components placed on a form at design time.

Figure 16.1 Decision support components at design time



		ShipVIA			
Terms	Country	DHL	Emery	FedEx	UPS
FOB	Algeria				$15,214.05
	America	$82,389.00	$20,321.75	$1,465.60	$66,475.85
	Canada	$10,152.00		$3,304.85	$48,988.40
	China	$67,448.65	$3,531.80	$787.80	$7,671.90
	France	$6,785.40		$20,108.00	$76,311.05

About crosstabs

Cross-tabulations, or crosstabs, are a way of presenting subsets of data so that relationships and trends are more visible. Table fields become the dimensions of the crosstab while field values define categories and summaries within a dimension.

You can use the decision support components to set up crosstabs in forms. *TDecisionGrid* shows data in a table, while *TDecisionGraph* charts it graphically. *TDecisionPivot* has buttons that make it easier to display and hide dimensions and move them between columns and rows.

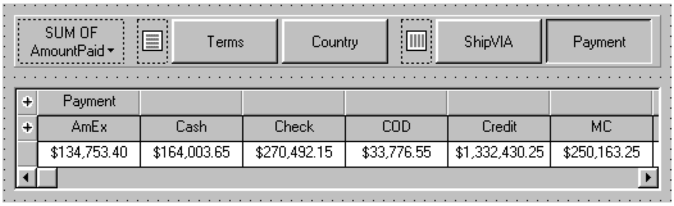
Crosstabs can be one-dimensional or multidimensional.

One-dimensional crosstabs

One-dimensional crosstabs show a summary row (or column) for the categories of a single dimension. For example, if Payment is the chosen column dimension and

Amount Paid is the summary category, the crosstab in Figure 16.2 shows the amount paid using each method.

Figure 16.2 One-dimensional crosstab



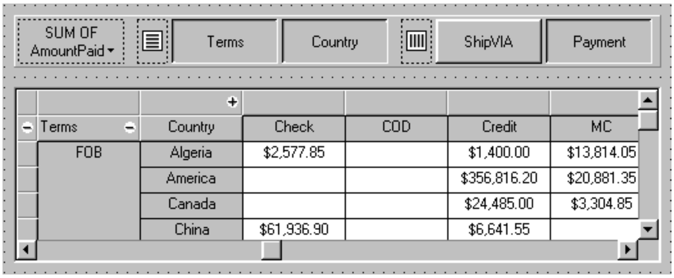
Payment					
AmEx	Cash	Check	COD	Credit	MC
$134,753.40	$164,003.65	$270,492.15	$33,776.55	$1,332,430.25	$250,163.25

Multidimensional crosstabs

Multidimensional crosstabs use additional dimensions for the rows and/or columns. For example, a two-dimensional crosstab could show amounts paid by payment method for each country.

A three-dimensional crosstab could show amounts paid by payment method and terms by country, as shown in Figure 16.3.

Figure 16.3 Three-dimensional crosstab



Terms	Country	Check	COD	Credit	MC
FOB	Algeria	$2,577.85		$1,400.00	$13,814.05
	America			$356,816.20	$20,881.35
	Canada			$24,485.00	$3,304.85
	China	$61,936.90		$6,641.55	

Guidelines for using decision support components

The decision support components listed on page 16-1 can be used together to present multidimensional data as tables and graphs. More than one grid or graph can be attached to each dataset. More than one instance of *TDecisionPivot* can be used to display the data from different perspectives at runtime.

To create a form with tables and graphs of multidimensional data, follow these steps:

1. Create a form.
2. Add these components to the form and use the Object Inspector to bind them as indicated:
   - A dataset, usually *TDecisionQuery* (for details, see “Creating decision datasets with the Decision Query editor” on page 16-6) or *TQuery*

- A decision cube, *TDecisionCube*, bound to the dataset by setting its *DataSet* property to the dataset's name
- A decision source, *TDecisionSource*, bound to the decision cube by setting its *DecisionCube* property to the decision cube's name

3 Add a decision pivot, *TDecisionPivot*, and bind it to the decision source with the Object Inspector by setting its *DecisionSource* property to the appropriate decision source name. The decision pivot is optional but useful; it lets the form developer and end users change the dimensions displayed in decision grids or decision graphs by pushing buttons.

In its default orientation, horizontal, buttons on the left side of the decision pivot apply to fields on the left side of the decision grid (rows); buttons on the right side apply to fields at the top of the decision grid (columns).

You can determine where the decision pivot's buttons appear by setting its *GroupLayout* property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default). For more information on decision pivot properties, see "Using decision pivots" on page 16-9.

4 Add one or more decision grids and graphs, bound to the decision source. For details, see "Creating and using decision grids" on page 16-10 and "Creating and using decision graphs" on page 16-13.

5 Use the Decision Query editor or *SQL* property of *TDecisionQuery* (or *TQuery*) to specify the tables, fields, and summaries to display in the grid or graph. The last field of the SQL SELECT should be the summary field. The other fields in the SELECT must be GROUP BY fields. For instructions, see "Creating decision datasets with the Decision Query editor" on page 16-6.

6 Set the *Active* property of the decision query (or alternate dataset component) to *True*.

7 Use the decision grid and graph to show and chart different data dimensions. See "Using decision grids" on page 16-11 and "Using decision graphs" on page 16-13 for instructions and suggestions.

For an illustration of all decision support components on a form, see Figure 16.1 on page 16-2.

Using datasets with decision support components

The only decision support component that binds directly to a dataset is the decision cube, *TDecisionCube. TDecisionCube* expects to receive data with groups and summaries defined by an SQL statement of an acceptable format. The GROUP BY phrase must contain the same non-summarized fields (and in the same order) as the SELECT phrase, and summary fields must be identified.

The decision query component, *TDecisionQuery*, is a specialized form of *TQuery*. You can use TDecisionQuery to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes (*TDecisionCube)*. You can also use an ordinary *TQuery* or other BDE-enabled dataset as a dataset for *TDecisionCube*, but the correct setup of the dataset and *TDecisionCube* are then the responsibility of the designer.

To work correctly with the decision cube, all projected fields in the dataset must either be dimensions or summaries. The summaries should be additive values (like sum or count), and should represent totals for each combination of dimension values. For maximum ease of setup, sums should be named “Sum...” in the dataset while counts should be named “Count...”.

The Decision Cube can pivot, subtotal, and drill-in correctly only for summaries whose cells are additive. (SUM and COUNT are additive, while AVERAGE, MAX, and MIN are not.) Build pivoting crosstab displays only for grids that contain only additive aggregators. If you are using non-additive aggregators, use a static decision grid that does not pivot, drill, or subtotal.

Since averages can be calculated using SUM divided by COUNT, a pivoting average is added automatically when SUM and COUNT dimensions for a field are included in the dataset. Use this type of average in preference to an average calculated using an AVERAGE statement.

Averages can also be calculated using COUNT(*). To use COUNT(*) to calculate averages, include a "COUNT(*) COUNTALL" selector in the query. If you use COUNT(*) to calculate averages, the single aggregator can be used for all fields. Use COUNT(*) only in cases where none of the fields being summarized include blank values, or where a COUNT aggregator is not available for every field.

Creating decision datasets with TQuery or TTable

If you use an ordinary *TQuery* component as a decision dataset, you must manually set up the SQL statement, taking care to supply a GROUP BY phrase which contains the same fields (and in the same order) as the SELECT phrase.

The SQL should look similar to this:

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

The ordering of the SELECT fields should match the ordering of the GROUP BY fields.

With *TTable*, you must supply information to the decision cube about which of the fields in the query are grouping fields, and which are summaries. To do this, Fill in the Dimension Type for each field in the *DimensionMap* of the Decision Cube. You must indicate whether each field is a dimension or a summary, and if a summary, you must provide the summary type. Since pivoting averages depend on SUM/COUNT calculations, you must also provide the base field name to allow the decision cube to match pairs of SUM and COUNT aggregators.

Creating decision datasets with the Decision Query editor

All data used by the decision support components passes through the decision cube, which accepts a specially formatted dataset most easily produced by an SQL query. See “Using datasets with decision support components” on page 16-4 for more information.

While both *TTable* and *TQuery* can be used as decision datasets, it is easier to use *TDecisionQuery*; the Decision Query editor supplied with it can be used to specify tables, fields, and summaries to appear in the decision cube and will help you set up the SELECT and GROUP BY portions of the SQL correctly.

To use the Decision Query editor:

1. Select the decision query component on the form, then right-click and choose Decision Query editor. The Decision Query editor dialog box appears.
2. Choose the database to use.
3. For single-table queries, click the Select Table button.

   For more complex queries involving multi-table joins, click the Query Builder button to display the SQL Builder or type the SQL statement into the edit box on the SQL tab page.
4. Return to the Decision Query editor dialog box.
5. In the Decision Query editor dialog box, select fields in the Available Fields list box and assign them to be either Dimensions or Summaries by clicking the appropriate right arrow button. As you add fields to the Summaries list, select from the menu displayed the type of summary to use: sum, count, or average.
6. By default, all fields and summaries defined in the *SQL* property of the decision query appear in the Active Dimensions and Active Summaries list boxes. To remove a dimension or summary, select it in the list and click the left arrow beside the list, or double-click the item to remove. To add it back, select it in the Available Fields list box and click the appropriate right arrow.

Once you define the contents of the decision cube, you can further manipulate dimension display with its *DimensionMap* property and the buttons of *TDecisionPivot*. For more information, see the next section, “Using decision cubes,” “Using decision sources” on page 16-9, and “Using decision pivots” on page 16-9.

Note When you use the Decision Query editor, the query is initially handled in ANSI-92 SQL syntax, then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays only ANSI standard SQL. The dialect translation is automatically assigned to the *TDecisionQuery*’s SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather then the SQL property.

Using decision cubes

The decision cube component, *TDecisionCube,* is a multidimensional data store that fetches its data from a dataset (typically a specially structured SQL statement entered through *TDecisionQuery* or *TQuery*). The data is stored in a form that makes its easy to pivot (that is, change the way in which the data is organized and summarized) without needing to run the query a second time.

Decision cube properties and events

The *DimensionMap* properties of *TDecisionCube* not only control which dimensions and summaries appear but also let you set date ranges and specify the maximum number of dimensions the decision cube may support. You can also indicate whether or not to display data during design. You can display names, (categories) values, subtotals, or data. Display of data at design time can be time consuming, depending on the data source.

When you click the ellipsis next to *DimensionMap* in the Object Inspector, the Decision Cube editor dialog box appears. You can use its pages and controls to set the *DimensionMap* properties.

The *OnRefresh* event fires whenever the decision cube cache is rebuilt. Developers can access the new dimension map and change it at that time to free up memory, change the maximum summaries or dimensions, and so on. *OnRefresh* is also useful if users access the Decision Cube editor; application code can respond to user changes at that time.

Using the Decision Cube editor

You can use the Decision Cube editor to set the *DimensionMap* properties of decision cubes. You can display the Decision Cube editor through the Object Inspector, as described in the previous section, or by right-clicking a decision cube on a form at design time and choosing Decision Cube editor.

The Decision Cube Editor dialog box has two tabs:

- Dimension Settings, used to activate or disable available dimensions, rename and reformat dimensions, put dimensions in a permanently drilled state, and set date ranges to display.
- Memory Control, used to set the maximum number of dimensions and summaries that can be active at one time, to display information about memory usage, and to determine the names and data that appear at design time.

Viewing and changing dimension settings

To view the dimension settings, display the Decision Cube editor and click the Dimension Settings tab. Then, select a dimension or summary in the Available Fields list. Its information appears in the boxes on the right side of the editor:

- To change the dimension or summary name that appears in the decision pivot, decision grid, or decision graph, enter a new name in the Display Name edit box.
- To determine whether the selected field is a dimension or summary, read the text in the Type edit box. If the dataset is a *TTable* component, you can use Type to specify whether the selected field is a dimension or summary.
- To disable or activate the selected dimension or summary, change the setting in the Active Type drop-down list box: Active, As Needed, or Inactive. Disabling a dimension or setting it to As Needed saves memory.
- To change the format of that dimension or summary, enter a format string in the Format edit box.
- To display that dimension or summary by Year, Quarter, or Month, change the setting in the Binning drop-down list box. Note that you can choose Set in the Binning list box to put the selected dimension or summary in a permanently "drilled down" state. This can be useful for saving memory when a dimension has many values. For more information, see "Decision support components and memory control" on page 16-19.
- To determine the starting value for ranges, or the drill-down value for a "Set" dimension, first choose the appropriate Grouping value in the Grouping drop-down, and then enter the starting range value or permanent drill-down value in the Initial Value drop-down list.

Setting the maximum available dimensions and summaries

To determine the maximum number of dimensions and summaries available for decision pivots, decision grids, and decision graphs bound to the selected decision cube, display the Decision Cube editor and click the Memory Control tab. Use the edit controls to adjust the current settings, if necessary. These settings help to control the amount of memory required by the decision cube. For more information, see "Decision support components and memory control" on page 16-19.

Viewing and changing design options

To determine how much information appears at design time, display the Decision Cube editor and click the Memory Control tab. Then, check the setting that indicates which names and data to display. Display of data or field names at design time can cause performance delays in some cases because of the time needed to fetch the data.

Using decision sources

The decision source component, *TDecisionSource,* defines the current pivot state of decision grids or decision graphs. Any two objects which use the same decision source also share pivot states.

Properties and events

The following are some special properties and events that control the appearance and behavior of decision sources:

- The *ControlType* property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections).
- The *SparseCols* and *SparseRows* properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed.
- *TDecisionSource* has the following events:
  - *OnLayoutChange* occurs when the user performs pivots or drill-downs that reorganize the data.
  - *OnNewDimensions* occurs when the data is completely altered, such as when the summary or dimension fields are altered.
  - *OnSummaryChange* occurs when the current summary is changed.
  - *OnStateChange* occurs when the Decision Cube activates or deactivates.
  - *OnBeforePivot* occurs when changes are committed but not yet reflected in the user interface. Developers have an opportunity to make changes, for example, in capacity or pivot state, before application users see the result of their previous action.
  - *OnAfterPivot* fires after a change in pivot state. Developers can capture information at that time.

Using decision pivots

The decision pivot component, *TDecisionPivot,* lets you open or close decision cube dimensions, or fields, by pressing buttons. When a row or column is opened by pressing a *TDecisionPivot* button, the corresponding dimension appears on the *TDecisionGrid* or *TDecisionGraph* component. When a dimension is closed, its detailed data doesn’t appear; it collapses into the totals of other dimensions. A dimension may also be in a “drilled” state, where only the summaries for a particular value of the dimension field appear.

You can also use the decision pivot to reorganize dimensions displayed in the decision grid and decision graph. Just drag a button to the row or column area or reorder buttons within the same area.

For illustrations of decision pivots at design time, see Figures 16.1, 16.2, and 16.3.

Decision pivot properties

The following are some special properties that control the appearance and behavior of decision pivots:

- The first properties listed for *TDecisionPivot* define its overall behavior and appearance. You might want to set *ButtonAutoSize* to *False* for *TDecisionPivot* to keep buttons from expanding and contracting as you adjust the size of the component.
- The *Groups* property of *TDecisionPivot* defines which dimension buttons appear. You can display the row, column, and summary selection button groups in any combination. Note that if you want more flexibility over the placement of these groups, you can place one *TDecisionPivot* on your form which contains only rows in one location, and a second which contains only columns in another location.
- Typically, *TDecisionPivot* is added above *TDecisionGrid*. In its default orientation, horizontal, buttons on the left side of *TDecisionPivot* apply to fields on the left side of *TDecisionGrid* (rows); buttons on the right side apply to fields at the top of *TDecisionGrid* (columns).
- You can determine where *TDecisionPivot*'s buttons appear by setting its *GroupLayout* property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default, described in the previous paragraph).

Creating and using decision grids

Decision grid components, *TDecisionGrid,* present cross-tabulated data in table form. These tables are also called crosstabs, described on page 16-2. Figure 16.1 on page 16-2 shows a decision grid on a form at design time.

Creating decision grids

To create a form with one or more tables of cross-tabulated data,

1. Follow steps 1–3 listed under “Guidelines for using decision support components” on page 16-3.
2. Add one or more decision grid components (*TDecisionGrid*) and bind them to the decision source, *TDecisionSource,* with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.

3 Continue with steps 5–7 listed under “Guidelines for using decision support components.”

For a description of what appears in the decision grid and how to use it, see “Using decision grids” on page 16-11.

To add a graph to the form, follow the instructions in “Creating decision graphs” on page 16-13.

Using decision grids

The decision grid component, *TDecisionGrid*, displays data from decision cubes (*TDecisionCube*) bound to decision sources (*TDecisionSource*).

By default, the grid appears with dimension fields at its left side and/or top, depending on the grouping instructions defined in the dataset. Categories, one for each data value, appear under each field. You can

- Open and close dimensions
- Reorganize, or pivot, rows and columns
- Drill down for detail
- Limit dimension selection to a single dimension for each axis

For more information about special properties and events of the decision grid, see “Decision grid properties” on page 16-12.

Opening and closing decision grid fields

If a plus sign (+) appears in a dimension or summary field, one or more fields to its right are closed (hidden). You can open additional fields and categories by clicking the sign. A minus sign (-) indicates a fully opened (expanded) field. When you click the sign, the field closes. This outlining feature can be disabled; see “Decision grid properties” on page 16-12 for details.

Reorganizing rows and columns in decision grids

You can drag row and column headings to new locations within the same axis or to the other axis. In this way, you can reorganize the grid and see the data from new perspectives as the data groupings change. This pivoting feature can be disabled; see “Decision grid properties” on page 16-12 for details.

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See “Using decision pivots” on page 16-9 for instructions.

Drilling down for detail in decision grids

You can drill down to see more detail in a dimension.

For example, if you right-click a category label (row heading) for a dimension with others collapsed beneath it, you can choose to drill down and only see data for that category. When a dimension is drilled, you do not see the category labels for that dimension displayed on the grid, since only the records for a single category value are being displayed. If you have a decision pivot on the form, it displays category values and lets you change to other values if you want.

To drill down into a dimension,

- Right-click a category label and choose Drill In To This Value, or
- Right-click a pivot button and choose Drilled In.

To make the complete dimension active again,

- Right-click the corresponding pivot button, or right-click the decision grid in the upper-left corner and select the dimension.

Limiting dimension selection in decision grids

You can change the *ControlType* property of the decision source to determine whether more than one dimension can be selected for each axis of the grid. For more information, see "Using decision sources" on page 16-9.

Decision grid properties

The decision grid component, *TDecisionGrid*, displays data from the *TDecisionCube* component bound to *TDecisionSource.* By default, data appears in a grid with category fields on the left side and top of the grid.

The following are some special properties that control the appearance and behavior of decision grids:

- *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the Object Inspector, then select a dimension. Its properties then appear in the Object Inspector: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines the color of category labels, *FieldName* displays the name of the active dimension, *Format* can hold any standard format for that data type, and *Subtotals* indicates whether to display subtotals for that dimension. With summary fields, these same properties are used to changed the appearance of the data that appears in the summary area of the grid. When you're through setting dimension properties, either click a component in the form or choose a component in the drop-down list box at the top of the Object Inspector.
- The *Options* property of *TDecisionGrid* lets you control display of grid lines (*cgGridLines = True*), enabling of outline features (collapse and expansion of dimensions with + and - indicators; *cgOutliner = True*), and enabling of drag-and-drop pivoting (*cgPivotable = True*).
- The *OnDecisionDrawCell* event of *TDecisionGrid* gives you a chance to change the appearance of each cell as it is drawn. The event passes the *String*, *Font*, and *Color* of the current cell as reference parameters. You are free to alter those parameters to achieve effects such as special colors for negative values. In addition to the *DrawState* which is passed by *TCustomGrid*, the event passes *TDecisionDrawState*, which can be used to determine what type of cell is being drawn. Further information about the cell can be fetched using the *Cells*, *CellValueArray*, or *CellDrawState* functions.

- The *OnDecisionExamineCell* event of *TDecisionGrid* lets you hook the right-click-on-event to data cells, and is intended to allow a program to display information (such as detail records) about that particular data cell. When the user right-clicks a data cell, the event is supplied with all the information which is was used to compose the data value, including the currently active summary value and a *ValueArray* of all the dimension values which were used to create the summary value.

Creating and using decision graphs

Decision graph components, *TDecisionGraph,* present cross-tabulated data in graphic form. Each decision graph shows the value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see page 16-2. For illustrations of decision graphs at design time, see Figure 16.1 on page 16-2 and Figure 16.4 on page 16-14.

Creating decision graphs

To create a form with one or more decision graphs,

1. Follow steps 1–3 listed under “Guidelines for using decision support components” on page 16-3.
2. Add one or more decision graph components (*TDecisionGraph*) and bind them to the decision source, *TDecisionSource,* with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.
3. Continue with steps 5–7 listed under “Guidelines for using decision support components.”
4. Finally, right-click the graph and choose Edit Chart to modify the appearance of the graph series. You can set template properties for each graph dimension, then set individual series properties to override these defaults. For details, see “Customizing decision graphs” on page 16-15.

For a description of what appears in the decision graph and how to use it, see the next section, “Using decision graphs.”

To add a decision grid—or crosstab table—to the form, follow the instructions in “Creating and using decision grids” on page 16-10.

Using decision graphs

The decision graph component, *TDecisionGraph*, displays fields from the decision source (*TDecisionSource*) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (*TDecisionPivot*).

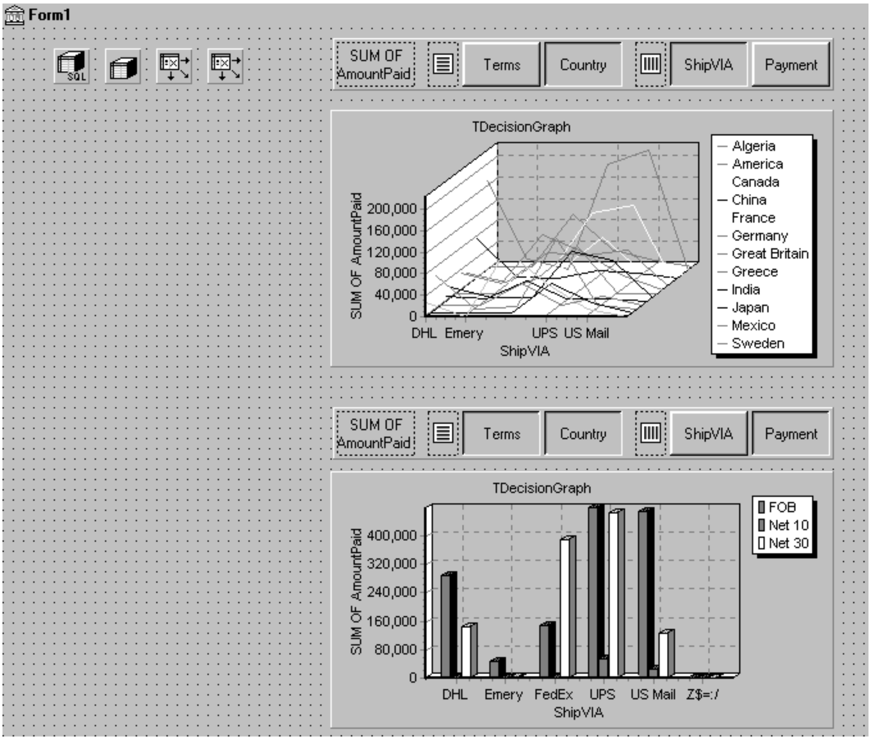
Graphed data comes from a specially formatted dataset such as *TDecisionQuery*. For an overview of how the decision support components handle and arrange this data, see page 16-1.

By default, the first row dimension appears as the x-axis and the first column dimension appears as the y-axis.

You can use decision graphs instead of or in addition to decision grids, which present cross-tabulated data in tabular form. Decision grids and decision graphs that are bound to the same decision source present the same data dimensions. To show different summary data for the same dimensions, you can bind more than one decision graph to the same decision source. To show different dimensions, bind decision graphs to different decision sources.

For example, in Figure 16.4 the first decision pivot and graph are bound to the first decision source and the second decision pivot and graph are bound to the second. So, each graph can show different dimensions.

Figure 16.4 Decision graphs bound to different decision sources



For more information about what appears in a decision graph, see the next section, “The decision graph display.”

To create a decision graph, see the previous section, “Creating decision graphs.”

For a discussion of decision graph properties and how to change the appearance and behavior of decision graphs, see “Customizing decision graphs” on page 16-15.

The decision graph display

By default, the decision graph plots summary values for categories in the first active row field (along the y-axis) against values in the first active column field (along the x-axis). Each graphed category appears as a separate series.

If only one dimension is selected—for example, by clicking only one *TDecisionPivot* button—only one series is graphed.

If you used a decision pivot, you can push its buttons to determine which decision cube fields (dimensions) are graphed. To exchange graph axes, drag the decision pivot dimension buttons from one side of the separator space to the other. If you have a one-dimensional graph with all buttons on one side of the separator space, you can use the Row or Column icon as a drop target for adding buttons to the other side of the separator and making the graph multidimensional.

If you only want one column and one row to be active at a time, you can set the *ControlType* property for *TDecisionSource* to *xtRadio*. Then, there can be only one active field at a time for each decision cube axis, and the decision pivot's functionality will correspond to the graph's behavior. *xtRadioEx* works the same as *xtRadio*, but does not allow the state where all row or all columns dimensions are closed.

When you have both a decision grid and graph connected to the same *TDecisionSource*, you'll probably want to set *ControlType* back to *xtCheck* to correspond to the more flexible behavior of *TDecisionGrid*.

Customizing decision graphs

The decision graph component, *TDecisionGraph,* displays fields from the decision source (*TDecisionSource*) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (*TDecisionPivot*). You can change the type, colors, marker types for line graphs, and many other properties of decision graphs.

To customize a graph,

1. Right-click it and choose Edit Chart. The Chart Editing dialog box appears.
2. Use the Chart page of the Chart Editing dialog box to view a list of visible series, select the series definition to use when two or more are available for the same series, change graph types for a template or series, and set overall graph properties.

   The Series list on the Chart page shows all decision cube dimensions (preceded by Template:) and currently visible categories. Each category, or series, is a separate object. You can:

   - Add or delete series derived from existing decision-graph series. Derived series can provide annotations for existing series or represent values calculated from other series.
   - Change the default graph type, and change the title of templates and series.

For a description of the other Chart page tabs, search for the following topic in online Help: “Chart page (Chart Editing dialog box).”

3 Use the Series page to establish dimension templates, then customize properties for each individual graph series.

By default, all series are graphed as bar graphs and up to 16 default colors are assigned. You can edit the template type and properties to create a new default. Then, as you pivot the decision source to different states, the template is used to dynamically create the series for each new state. For template details, see “Setting decision graph template defaults” on page 16-16.

To customize individual series, follow the instructions in “Customizing decision graph series” on page 16-17.

For a description of each Series page tab, search for the following topic in online Help: “Series page (Chart Editing dialog box).”

Setting decision graph template defaults

Decision graphs display the values from two dimensions of the decision cube: one dimension is displayed as an axis of the graph, and the other is used to create a set of series. The template for that dimension provides default properties for those series (such as whether the series are bar, line, area, and so on). As users pivot from one state to another, any required series for the dimension are created using the series type and other defaults specified in the template.

A separate template is provided for cases where users pivot to a state where only one dimension is active. A one-dimensional state is often represented with a pie chart, so a separate template is provided for this case.

You can

- Change the default graph type.
- Change other graph template properties.
- View and set overall graph properties.

Changing the default decision graph type

To change the default graph type,

1 Select a template in the Series list on the Chart page of the Chart Editing dialog box.

2 Click the Change button.

3 Select a new type and close the Gallery dialog box.

Changing other decision graph template properties

To change color or other properties of a template,

1 Select the Series page at the top of the Chart Editing dialog box.

2 Choose a template in the drop-down list at the top of the page.

3 Choose the appropriate property tab and select settings.

Viewing overall decision graph properties

To view and set decision graph properties other than type and series,

1. Select the Chart page at the top of the Chart Editing dialog box.
2. Choose the appropriate property tab and select settings.

Customizing decision graph series

The templates supply many defaults for each decision cube dimension, such as graph type and how series are displayed. Other defaults, such as series color, are defined by *TDecisionGraph*. If you want you can override the defaults for each series.

The templates are intended for use when you want the program to create the series for categories as they are needed, and discard them when they are no longer needed. If you want, you can set up custom series for specific category values. To do this, pivot the graph so its current display has a series for the category you want to customize. When the series is displayed on the graph, you can use the Chart editor to

- Change the graph type.
- Change other series properties.
- Save specific graph series that you have customized.

To define series templates and set overall graph defaults, see “Setting decision graph template defaults” on page 16-16.

Changing the series graph type

By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See “Changing the default decision graph type” on page 16-16 for instructions.

To change the graph type for a single series,

1. Select a series in the Series list on the Chart page of the Chart editor.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.
4. Check the Save Series check box.

Changing other decision graph series properties

To change color or other properties of a decision graph series,

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a series in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.
4. Check the Save Series check box.

Saving decision graph series settings

By default, only settings for templates are saved at design time. Changes made to specific series are only saved if the Save box is checked for that series in the Chart Editing dialog box.

Saving series can be memory intensive, so if you don't need to save them you can uncheck the Save box.

Decision support components at runtime

At runtime, users can perform many operations by left-clicking, right-clicking, and dragging visible decision support components. These operations, discussed earlier in this chapter, are summarized below.

Decision pivots at runtime

Users can:

- Left-click the summary button at the left end of the decision pivot to display a list of available summaries. They can use this list to change the summary data displayed in decision grids and decision graphs.
- Right-click a dimension button and choose to:
  - Move it from the row area to the column area or the reverse.
  - Drill In to display detail data.
- Left-click a dimension button following the Drill In command and choose:
  - Open Dimension to move back to the top level of that dimension.
  - All Values to toggle between displaying just summaries and summaries plus all other values in decision grids.
  - From a list of available categories for that dimension, a category to drill into for detail values.
- Left-click a dimension button to open or close that dimension.
- Drag and drop dimension buttons from the row area to the column area and the reverse; they can drop them next to existing buttons in that area or onto the row or column icon.

Decision grids at runtime

Users can:

- Right-click within the decision grid and choose to:
  - Toggle subtotals on and off for individual data groups, for all values of a dimension, or for the whole grid.
  - Display the Decision Cube editor, described on page 16-7.
  - Toggle dimensions and summaries open and closed.

- Click + and – within the row and column headings to open and close dimensions.
- Drag and drop dimensions from rows to columns and the reverse.

Decision graphs at runtime

Users can drag from side to side or up and down in the graph grid area to scroll through off-screen categories and values.

Decision support components and memory control

When a dimension or summary is loaded into the decision cube, it takes up memory. Adding a new summary increases memory consumption linearly: that is, a decision cube with two summaries uses twice as much memory as the same cube with only one summary, a decision cube with three summaries uses three times as much memory as the same cube with one summary, and so on. Memory consumption for dimensions increases more quickly. Adding a dimension with 10 values increases memory consumption by a factor of 10. Adding a dimension with 100 values increases memory consumption 100 times. Thus adding dimensions to a decision cube can have a dramatic effect on memory use, and can quickly lead to performance problems. This effect is especially pronounced when adding dimensions that have many values.

The decision support components have a number of settings to help you control how and when memory is used. For more information on the properties and techniques mentioned here, look up *TDecisionCube* in the online Help.

Setting maximum dimensions, summaries, and cells

The decision cube's *MaxDimensions* and *MaxSummaries* properties can be used with the *CubeDim.ActiveFlag* property to control how many dimensions and summaries can be loaded at a time. You can set the maximum values on the Cube Capacity page of the Decision Cube editor to place some overall control on how many dimensions or summaries can be brought into memory at the same time.

Limiting the number of dimensions or summaries provides a rough limit on the amount of memory used by the decision cube. However, it does not distinguish between dimensions with many values and those with only a few. For greater control of the absolute memory demands of the decision cube, you can also limit the number of cells in the cube. Set the maximum number of cells on the Cube Capacity page of the Decision Cube editor.

Setting dimension state

The *ActiveFlag* property controls which dimensions get loaded. You can set this property on the Dimension Settings tab of the Decision Cube editor using the Activity Type control. When this control is set to *Active*, the dimension is loaded unconditionally, and will always take up space. Note that the number of dimensions in this state must always be less than *MaxDimensions*, and the number of summaries set to *Active* must be less than *MaxSummaries*. You should set a dimension or summary to *Active* only when it is critical that it be available at all times. An *Active* setting decreases the ability of the cube to manage the available memory.

When *ActiveFlag* is set to *AsNeeded*, a dimension or summary is loaded only if it can be loaded without exceeding the *MaxDimensions*, *MaxSummaries,* or *MaxCells* limit. The decision cube will swap dimensions and summaries that are marked *AsNeeded* in and out of memory to keep within the limits imposed by *MaxCells*, *MaxDimensions*, and *MaxSummaries*. Thus, a dimension or summary may not be loaded in memory if it is not currently being used. Setting dimensions that are not used frequently to *AsNeeded* results in better loading and pivoting performance, although there will be a time delay to access dimensions which are not currently loaded.

Using paged dimensions

When Binning is set to Set on the Dimension Settings tab of the Decision cube editor and Start Value is not NULL, the dimension is said to be “paged,” or “permanently drilled down.” You can access data for just a single value of that dimension at a time, although you can programmatically access a series of values sequentially. Such a dimension may not be pivoted or opened.

It is extremely memory intensive to include dimensional data for dimensions that have very large numbers of values. By making such dimensions paged, you can display summary information for one value at a time. Information is usually easier to read when displayed this way, and memory consumption is much easier to manage.

Chapter

17

Connecting to databases

Most dataset components can connect directly to a database server. Once connected, the dataset communicates with the server automatically. When you open the dataset, it populates itself with data from the server, and when you post records, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection.

Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table lists these data access mechanisms and the associated connection components:

Table 17.1 Database connection components

Data access mechanism	Connection component
The Borland Database Engine (BDE).	TDatabase
ActiveX Data Objects (ADO).	TADOConnection
dbExpress.	TSQLConnection
InterBase Express.	TIBDatabase

Note For a discussion of some pros and cons of each of these mechanisms, see "Using databases" on page 14-1.

The connection component provides all the information necessary to establish a database connection. This information is different for each type of connection component:

- For information about describing a BDE-based connection, see "Identifying the database" on page 20-13.
- For information about describing an ADO-based connection, see "Connecting to a data store using TADOConnection" on page 21-3
- For information about describing a dbExpress connection, see "Setting up TSQLConnection" on page 22-3

- For information about describing an InterBase Express connection, see the online help for *TIBDatabase*.

Although each type of dataset uses a different connection component, they are all descendants of *TCustomConnection*. They all perform many of the same tasks and surface many of the same properties, methods, and events. This chapter discusses many of these common tasks.

Using implicit connections

No matter what data access mechanism you are using, you can always create the connection component explicitly and use it to manage the connection to and communication with a database server. For BDE-enabled and ADO-based datasets, you also have the option of describing the database connection through properties of the dataset and letting the dataset generate an implicit connection. For BDE-enabled datasets, you specify an implicit connection using the *DatabaseName* property. For ADO-based datasets, you use the *ConnectionString* property.

When using an implicit connection, you do not need to explicitly create a connection component. This can simplify your application development, and the default connection you specify can cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own connection components to tune each database connection to your application's needs. Explicit connection components give you greater control. For example, you need to access the connection component to perform the following tasks:

- Customize database server login support. (Implicit connections display a default login dialog to prompt the user for a user name and password.)
- Control transactions and specify transaction isolation levels.
- Execute SQL commands on the server without using a dataset.
- Perform actions on all open datasets that are connected to the same database.

In addition, if you have multiple datasets that all use the same server, it can be easier to use an connection component, so that you only have to specify the server to use in one place. That way, if you later change the server, you do not need to update several dataset components: only the connection component.

Controlling connections

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server. Each type of connection component surfaces a different set of properties to let you identify the server. In general, however, they all provide a way for you to name the server you want and supply a set of connection parameters that control how the connection is formed. Connection parameters vary from server to server. They can include information such as user name and password, the maximum size of BLOB fields, SQL roles, and so on.

Once you have identified the desired server and any connection parameters, you can use the connection component to explicitly open or close a connection. The connection component generates events when it opens or closes a connection that you can use to customize the response of your application to changes in the database connection.

Connecting to a database server

There are two ways to connect to a database server using a connection component:

- Call the *Open* method.
- Set the *Connected* property to *True*.

Calling the *Open* method sets *Connected* to *True*.

Note When a connection component is not connected to a server and an application attempts to open one of its associated datasets, the dataset automatically calls the connection component's *Open* method.

When you set *Connected* to *True*, the connection component first generates a *BeforeConnect* event, where you can perform any initialization. For example, you can use this event to alter connection parameters.

After the *BeforeConnect* event, the connection component may display a default login dialog, depending on how you choose to control server login. It then passes the user name and password to the driver, opening a connection.

Once the connection is open, the connection component generates an *AfterConnect* event, where you can perform any tasks that require an open connection.

Note Some connection components generate additional events as well when establishing a connection.

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, the connection component drops the connection. Some connection components surface a *KeepConnection* property that allows the connection to remain open even if all the datasets that use it are closed. If *KeepConnection* is *True*, the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting *KeepConnection* to *True* reduces network traffic and speeds up the application. If *KeepConnection* is *False,* the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

Disconnecting from a database server

There are two ways to disconnect a server using a connection component:

- Set the *Connected* property to *False*.
- Call the *Close* method.

Calling *Close* sets *Connected* to *False*.

When *Connected* is set to *False*, the connection component generates a *BeforeDisconnect* event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, the connection component closes all open datasets and disconnects from the server.

Finally, the connection component generates an *AfterDisconnect* event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

Note Calling *Close* or setting *Connected* to *False* disconnects from a database server even if the connection component has a *KeepConnection* property that is *True*.

Controlling server login

Most remote database servers include security features to prohibit unauthorized access. Usually, the server requires a user name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

- Let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt* property of the connection component to *True* (the default) and add DBLogDlg to the uses clause of the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.
- Supply the login information before the login attempt. Each type of connection component uses a different mechanism for specifying the user name and password:
  - For BDE, dbExpress, and InterBase express datasets, the user name and password connection parameters can be accessed through the *Params* property. (For BDE datasets, the parameter values can also be associated with a BDE alias, while for dbExpress datasets, they can also be associated with a connection name).
  - For ADO datasets, the user name and password can be included in the *ConnectionString* property (or provided as parameters to the *Open* method).

  If you specify the user name and password before the server requests them, be sure to set the *LoginPrompt* to *False*, so that the default login dialog does not appear. For example, the following code sets the user name and password on a SQL connection component in the *BeforeConnect* event handler, decrypting an encrypted password that is associated with the current connection name:

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
```

```
    if LoginPrompt = False then
    begin
      Params.Values['User_Name'] := 'SYSDBA';
      Params.Values['Password'] := Decrypt(Params.Values['Password']);
    end;
  end;
end;
```

Note that setting the user name and password at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. This still leaves them easy to find, compromising server security.

- Provide your own custom handling for the login event. The connection component generates an event when it needs the user name and password.
  - For *TDatabase*, *TSQLConnection*, and *TIBDatabase*, this is an *OnLogin* event. The event handler has two parameters, the connection component, and a local copy of the user name and password parameters in a string list. (*TSQLConnection* includes the database parameter as well). You must set the *LoginPrompt* property to *True* for this event to occur. Having a *LoginPrompt* value of *False* and assigning a handler for the *OnLogin* event creates a situation where it is impossible to log in to the database because the default dialog does not appear and the *OnLogin* event handler never executes.
  - For *TADOConnection*, the event is an *OnWillConnect* event. The event handler has five parameters, the connection component and four parameters that return values to influence the connection (including two for user name and password). This event always occurs, regardless of the value of *LoginPrompt*.

  Write an event handler for the event in which you set the login parameters. Here is an example where the values for the USER NAME and PASSWORD parameters are provided from a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*)

```
procedure TForm1.Database1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;
```

  As with the other methods of providing login parameters, when writing an *OnLogin* or *OnWillConnect* event handler, avoid hard coding the password in your application code. It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

Managing transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database. Most databases provide their own transaction management model, although some have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, connection components provide a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (The BDE also provides limited transaction support for local tables with no server transaction support. When not using the BDE, trying to start transactions on a database that does not support them causes connection components to raise an exception.)

Warning When a dataset provider component applies updates, it implicitly generates transactions for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

Starting a transaction

When you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or (in the case of overlapping transactions) until another transaction is started. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

While the transaction is in process, your view of the data in database tables is determined by your transaction isolation level. For information about transaction isolation levels, see “Specifying the transaction isolation level” on page 17-9.

For *TADOConnection*, start a transaction by calling the *BeginTrans* method:

```
Level := ADOConnection1.BeginTrans;
```

BeginTrans returns the level of nesting for the transaction that started. A nested transaction is one that is nested within another, parent, transaction. After the server starts the transaction, the ADO connection receives an *OnBeginTransComplete* event.

For *TDatabase,* use the *StartTransaction*method instead. *TDataBase* does not support nested or overlapped transactions: If you call a *TDatabase* component’s *StartTransaction* method while another transaction is underway, it raises an exception. To avoid calling *StartTransaction*, you can check the *InTransaction* property:

```
if not Database1.InTransaction then
  Database1.StartTransaction;
```

TSQLConnection also uses the *StartTransaction*method, but it uses a version that gives you a lot more control. Specifically, *StartTransaction* takes a transaction descriptor, which lets you manage multiple simultaneous transactions and specify the transaction isolation level on a per-transaction basis. (For more information on transaction levels, see "Specifying the transaction isolation level" on page 17-9.) In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested (as they can be when using ADO) or they can be overlapped.

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

By default, with overlapped transactions, the first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. If you are using *TSQLConnection* with an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its *TransactionLevel* property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

Note Unlike *TADOConnection*, *TSQLConnection* and *TDatabase* do not receive any events when the transactions starts.

InterBase express offers you even more control than *TSQLConnection* by using a separate transaction component rather than starting transactions using the connection component. You can, however, use *TIBDatabase* to start a default transaction:

```
if not IBDatabase1.DefaultTransaction.InTransaction then
  IBDatabase1.DefaultTransaction.StartTransaction;
```

You can have overlapped transactions by using two separate transaction components. Each transaction component has a set of parameters that let you configure the transaction. These let you specify the transaction isolation level, as well as other properties of the transaction.

Ending a transaction

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

Ending a successful transaction

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by committing the transaction. For *TDatabase*, you commit a transaction using the *Commit* method:

```
MyOracleConnection.Commit;
```

For *TSQLConnection*, you also use the *Commit* method, but you must specify which transaction you are committing by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Commit(TD);
```

For *TIBDatabase*, you commit a transaction object using its *Commit* method:

```
IBDatabase1.DefaultTransaction.Commit;
```

For *TADOConnection*, you commit a transaction using the *CommitTrans* method:

```
ADOConnection1.CommitTrans;
```

Note It is possible for a nested transaction to be committed, only to have the changes rolled back later if the parent transaction is rolled back.

After the transaction is successfully committed, an ADO connection component receives an *OnCommitTransComplete* event. Other connection components do not receive any similar events.

A call to commit the current transaction is usually attempted in a **try...except** statement. That way, if the transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

Ending an unsuccessful transaction

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. Discarding these changes is called rolling back the transaction.

For *TDatabase*, you roll back a transaction by calling the *Rollback* method:

```
MyOracleConnection.Rollback;
```

For *TSQLConnection*, you also use the *Rollback* method, but you must specify which transaction you are rolling back by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Rollback(TD);
```

For *TIBDatabase*, you roll back a transaction object by calling its *Rollback* method:

```
IBDatabase1.DefaultTransaction.Rollback;
```

For *TADOConnection*, you roll back a transaction by calling the *RollbackTrans* method:

```
ADOConnection1.RollbackTrans;
```

After the transaction is successfully rolled back, an ADO connection component receives an *OnRollbackTransComplete* event. Other connection components do not receive any similar events.

A call to roll back the current transaction usually occurs in

- Exception handling code when you can't recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Specifying the transaction isolation level

Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Each server type supports a different set of possible transaction isolation levels. There are three possible transaction isolation levels:

- *DirtyRead*: When the isolation level is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle, Sybase, MS-SQL, and InterBase).
- *ReadCommitted*: When the isolation level is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading. This level is available for all transactions except local transactions managed by the BDE.
- *RepeatableRead*: When the isolation level is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions. This level is not available on some servers, such as Sybase and MS-SQL and is unavailable on local transactions managed by the BDE.

In addition, *TSQLConnection* lets you specify database-specific custom isolation levels. Custom isolation levels are defined by the *dbExpress* driver. See you driver documentation for details.

Note For a detailed description of how each isolation level is implemented, see your server documentation.

TDatabase and *TADOConnection* let you specify the transaction isolation level by setting the *TransIsolation* property. When you set *TransIsolation* to a value that is not supported by the database server, you get the next highest level of isolation (if available). If there is no higher level available, the connection component raises an exception when you try to start a transaction.

When using *TSQLConnection*, transaction isolation level is controlled by the *IsolationLevel* field of the transaction descriptor.

When using InterBase express, transaction isolation level is controlled by a transaction parameter.

Sending commands to the server

All database connection components except *TIBDatabase* let you execute SQL statements on the associated server by calling the *Execute* method. Although *Execute* can return a cursor when the statement is a SELECT statement, this use is not recommended. The preferred method for executing statements that return data is to use a dataset.

The *Execute* method is very convenient for executing simple SQL statements that do not return any records. Such statements include Data Definition Language (DDL) statements, which operate on or create a database's metadata, such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Some Data Manipulation Language (DML) SQL statements also do not return a result set. The DML statements that perform an action on data but do not return a result set are: INSERT, DELETE, and UPDATE.

The syntax for the *Execute* method varies with the connection type:

- For *TDatabase*, *Execute* takes four parameters: a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, a boolean that indicates whether the statement should be cached because you will call it again, and a pointer to a BDE cursor that can be returned (It is recommended that you pass nil).
- For *TADOConnection*, there are two versions of *Execute.* The first takes a WideString that specifies the SQL statement and a second parameter that specifies a set of options that control whether the statement is executed asynchronously and whether it returns any records. This first syntax returns an interface for the returned records. The second syntax takes a WideString that specifies the SQL statement, a second parameter that returns the number of records affected when the statement executes, and a third that specifies options such as whether the statement executes asynchronously. Note that neither syntax provides for passing parameters.
- For *TSQLConnection*, *Execute* takes three parameters: a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, and a pointer that can receive a *TCustomSQLDataSet* that is created to return records.

Note *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters on a *TSQLConnection* component:

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

To use parameters, you must create a *TParams* object. For each parameter value, use the *TParams.CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which uses *TDatabase* to execute an INSERT statement. The INSERT statement has a single parameter named :*StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of "CA" for that parameter.

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    Database1.Execute(SQLstmt, stmtParams, False, nil);
  finally
    stmtParams.Free;
  end;
end;
```

If the SQL statement includes a parameter but you do not supply a *TParam* object to provide its value, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

Working with associated datasets

All database connection components maintain a list of all datasets that use them to connect to a database. A connection component uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific connection component to connect to a particular database.

Closing all datasets without disconnecting from the server

The connection component automatically closes all datasets when you close its connection. There may be times, however, when you want to close all datasets without disconnecting from the database server.

To close all open datasets without disconnecting from a server, you can use the *CloseDataSets* method.

For *TADOConnection* and *TIBDatabase*, calling *CloseDataSets* always leaves the connection open. For *TDatabase* and *TSQLConnection*, you must also set the *KeepConnection* property to *True*.

Iterating through the associated datasets

To perform any actions (other than closing them all) on all the datasets that use a connection component, use the *DataSets* and *DataSetCount* properties. *DataSets* is an indexed array of all datasets that are linked to the connection component. For all connection components except *TADOConnection*, this list includes only the active datasets. *TADOConnection* lists the inactive datasets as well. *DataSetCount* is the number of datasets in this array.

Note When you use a specialized client dataset to cache updates (as opposed to the generic client dataset, *TClientDataSet*), the *DataSets* property lists the internal dataset owned by the client dataset, not the client dataset itself.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

```
var
  I: Integer;
begin
  with MyDBConnection do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
```

Note *TADOConnection* supports command objects as well as datasets. You can iterate through these much like you iterate through the datasets, by using the *Commands* and *CommandCount* properties.

Obtaining metadata

All database connection components can retrieve lists of metadata on the database server, although they vary in the types of metadata they retrieve. The methods that retrieve metadata fill a string list with the names of various entities available on the server. You can then use this information, for example, to let your users dynamically select a table at runtime.

You can use a *TADOConnection* component to retrieve metadata about the tables and stored procedures available on the ADO data store. You can then use this information, for example, to let your users dynamically select a table or stored procedure at runtime.

Listing available tables

The *GetTableNames* method copies a list of table names to an already-existing string list object. This can be used, for example, to fill a list box with table names that the user can then use to choose a table to open. The following line fills a listbox with the names of all tables on the database:

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

GetTableNames has two parameters: the string list to fill with table names, and a boolean that indicates whether the list should include system tables, or ordinary tables. Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.

Note For most database connection components, *GetTableNames* returns a list of all available non-system tables when the second parameter is *False*. For *TSQLConnection*, however, you have more control over what type is added to the list when you are not fetching only the names of system tables. When using *TSQLConnection*, the types of names added to the list are controlled by the *TableScope* property. *TableScope* indicates whether the list should contain any or all of the following: ordinary tables, system tables, synonyms, and views.

Listing the fields in a table

The *GetFieldNames* method fills an existing string list with the names of all fields (columns) in a specified table. *GetFieldNames* takes two parameters, the name of the table for which you want to list the fields, and an existing string list to be filled with field names:

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

Listing available stored procedures

To get a listing of all of the stored procedures contained in the database, use the *GetProcedureNames* method. This method takes a single parameter: an already-existing string list to fill:

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

Note *GetProcedureNames* is only available for *TADOConnection* and *TSQLConnection*.

Listing available indexes

To get a listing of all indexes defined for a specific table, use the *GetIndexNames* method. This method takes two parameters: the table whose indexes you want, and an already-existing string list to fill:

```
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
```

Note *GetIndexNames* is only available for *TSQLConnection*, although most table-type datasets have an equivalent method.

Listing stored procedure parameters

To get a list of all parameters defined for a specific stored procedure, use the *GetProcedureParams* method. *GetProcedureParams* fills a *TList* object with pointers to parameter description records, where each record describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on.

GetProcedureParams takes two parameters: the name of the stored procedure, and an already-existing *TList* object to fill:

```
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
```

You can convert the parameter descriptions that are added to the list into the more familiar *TParams* object by calling the global *LoadParamListItems*procedure. Because *GetProcedureParams* dynamically allocates the individual records, your application must free them when it is finished with the information. The global *FreeProcParams* routine can do this for you.

Note *GetProcedureParams* is only available for *TSQLConnection*.

Chapter

18

Understanding datasets

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure.

All dataset objects that you use in your database applications descend from *TDataSet*, and they inherit data fields, properties, events, and methods from this class. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you use in your database applications. You need to understand this shared functionality to use any dataset object.

TDataSet is a virtualized dataset, meaning that many of its properties and methods are **virtual** or **abstract**. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of the built-in *TDataSet* descendants and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its **abstract** methods.

TDataSet defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For information about *TField* components, see Chapter 19, “Working with field components.”

This chapter describes how to use the common database funtionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* dependants implement them. In particular, unidirectional datasets implement only a limited subset.

Using TDataSet descendants

TDataSet has several immediate descendants, each of which corresponds to a different data access mechanism. You do not work directly with any of these descendants. Rather, each descendant introduces the properties and methods for using a particular data access mechanism. These properties and methods are then exposed by descendant classes that are adapted to different types of server data. The immediate descendants of *TDataSet* include

- *TBDEDataSet*, which uses the Borland Database Engine (BDE) to communicate with the database server. The *TBDEDataSet* descendants you use are *TTable*, *TQuery*, *TStoredProc*, and *TNestedTable*. The unique features of BDE-enabled datasets are described in Chapter 20, “Using the Borland Database Engine”.
- *TCustomADODataSet*, which uses ActiveX Data Objects (ADO) to communicate with an OLEDB data store. The *TCustomADODataSet* descendants you use are *TADODataSet*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. The unique features of ADO-based datasets are described in Chapter 21, “Working with ADO components”.
- *TCustomSQLDataSet*, which uses dbExpress to communicate with a database server. The *TCustomSQLDataSet* descendants you use are *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*. The unique features of dbExpress datasets are described in Chapter 22, “Using unidirectional datasets”.
- *TIBCustomDataSet*, which communicates directly with an InterBase database server. The *TIBCustomDataSet* descendants you use are *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TIBStoredProc*.
- *TCustomClientDataSet*, which represents the data from another dataset component or the data from a dedicated file on disk. The *TCustomClientDataSet* descendants you use are *TClientDataSet*, which can connect to an external (source) dataset, and the client datasets that are specialized to a particular data access mechanism (*TBDEClientDataSet*, *TSQLClientDataSet*, and *TIBClientDataSet*), which use an internal source dataset. The unique features of client datasets are described in Chapter 23, “Using client datasets”.

Some pros and cons of the various data access mechanisms employed by these *TDataSet* descendants are described in “Using databases” on page 14-1.

In addition to the built-in datasets, you can create your own custom *TDataSet* descendants — for example to supply data from a process other than a database server, such as a spreadsheet. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see Chapter 40, “Overview of component creation.”

Although each *TDataSet* descendant has its own unique properties and methods, some of the properties and methods introduced by descendant classes are the same as those introduced by other descendant classes that use another data access mechanism. For example, there are similarities between the “table” components (*TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*). For information about the commonalities introduced by *TDataSet* descendants, see “Types of datasets” on page 18-23.

Determining dataset states

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset’s read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Table 18.1 Values for the dataset State property

Value	State	Meaning
dsInactive	Inactive	DataSet closed. Its data is unavailable.
dsBrowse	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	DataSet open. The current row can be modified. (not supported on unidirectional datasets)
dsInsert	Insert	DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets)
dsSetKey	SetKey	DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations. (not supported by all datasets)
dsCalcFields	CalcFields	DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsCurValue	CurValue	DataSet open. Indicates that the CurValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsNewValue	NewValue	DataSet open. Indicates that the NewValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsOldValue	OldValue	DataSet open. Indicates that the OldValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsFilter	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets)
dsBlockRead	Block Read	DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes.

Table 18.1 Values for the dataset State property (continued)

Value	State	Meaning
dsInternalCalc	Internal Calc	DataSet open. An *OnCalcFields* event is underway for calculated values that are stored with the record. (client datasets only)
dsOpening	Opening	DataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching.

Typically, an application checks the dataset state to determine when to perform certain tasks. For example, you might check for the *dsEdit* or *dsInsert* state to ascertain whether you need to post updates.

Note Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see "Responding to changes mediated by the data source" on page 15-4.

Opening and closing datasets

To read or write data in a dataset, an application must first open it. You can open a dataset in two ways,

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at runtime:

```
CustTable.Active := True;
```

- Call the *Open* method for the dataset at runtime,

```
CustQuery.Open;
```

When you open the dataset, the dataset first receives a *BeforeOpen* event, then it opens a cursor, populating itself with data, and finally, it receives an *AfterOpen* event.

The newly-opened dataset is in browse mode, which means your application can read the data and navigate through it.

You can close a dataset in two ways,

- Set the *Active* property of the dataset to *False*, either at design time in the Object Inspector, or in code at runtime,

```
CustQuery.Active := False;
```

- Call the *Close* method for the dataset at runtime,

```
CustTable.Close;
```

Just as the dataset receives *BeforeOpen* and *AfterOpen* events when you open it, it receives a *BeforeClose* and *AfterClose* event when you close it. handlers that respond to the *Close* method for a dataset. You can use these events, for example, to prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { save the changes }
      mrNo:     CustTable.Cancel; { abandon the changes}
      mrCancel: Abort;            { abort closing the dataset }
    end;
  end;
end;
```

Note You may need to close a dataset when you want to change certain of its properties, such as *TableName* on a *TTable* component. When you reopen the dataset, the new property value takes effect.

Navigating datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Table 18.2 Navigational methods of datasets

Method	Moves the cursor to
First	The first row in a dataset.
Last	The last row in a dataset. (not available for unidirectional datasets)
Next	The next row in a dataset.
Prior	The previous row in a dataset. (not available for unidirectional datasets)
MoveBy	A specified number of rows forward or back in a dataset.

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime. For information about the navigator component, see “Navigating and manipulating records” on page 15-28.

Whenever you change the current record using one of these methods (or by other methods that navigate based on a search criterion), the dataset receives two events: *BeforeScroll* (before leaving the current record) and *AfterScroll* (after arriving at the new record). You can use these events to update your user interface (for example, to update a status bar that indicates information about the current record).

TDataSet also defines two boolean properties that provide useful information when iterating through the records in a dataset.

Table 18.3 Navigational properties of datasets

Property	Description
Bof (Beginning-of-file)	*True*: the cursor is at the first row in the dataset.
	False: the cursor is not known to be at the first row in the dataset
Eof (End-of-file)	*True*: the cursor is at the last row in the dataset.
	False: the cursor is not known to be at the first row in the dataset

Using the First and Last methods

The *First* method moves the cursor to the first row in a dataset and sets the *Bof* property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable.First;
```

The *Last* method moves the cursor to the last row in a dataset and sets the *Eof* property to *True*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable.Last;
```

Note The *Last* method raises an exception in unidirectional datasets.

Tip While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons call the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see “Navigating and manipulating records” on page 15-28.

Using the Next and Prior methods

The *Next* method moves the cursor forward one row in the dataset and sets the *Bof* property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable.Next;
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* to *False* if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
CustTable.Prior;
```

Note The *Prior* method raises an exception in unidirectional datasets.

Using the MoveBy method

MoveBy lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the *Bof* and *Eof* properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

Note *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

MoveBy returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable.MoveBy(-2);
```

Note If your application uses *MoveBy* in a multi-user database environment, keep in mind that datasets are fluid. A record that was five records back a moment ago may now be four, six, or even an unknown number of records back if several users are simultaneously accessing the database and changing its data.

Using the Eof and Bof properties

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful when you want to iterate through all records in a dataset.

Eof

When *Eof* is *True*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *True* when an application

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.
- Calls *SetRange* on an empty range or dataset.

Eof is set to *False* in all other cases; you should assume *Eof* is *False* unless one of the conditions above is met *and* you test the property directly.

Eof is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call *First*) *Eof* is

False. To iterate through the dataset a record at a time, create a loop that steps through each record by calling *Next*, and terminates when *Eof* is *True*. *Eof* remains *False* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets Eof False }
  while not CustTable.Eof do { Cycle until Eof is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Next; { Eof False on success; Eof True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

Tip This example also shows how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

Bof

When *Bof* is *True*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *True* when an application

- Opens a dataset.
- Calls a dataset's *First* method.
- Calls a dataset's *Prior* method, and the method fails (because the cursor is currently at the first row in the dataset.
- Calls *SetRange* on an empty range or dataset.

Bof is set to *False* in all other cases; you should assume *Bof* is *False* unless one of the conditions above is met *and* you test the property directly.

Like *Eof*, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.Bof do { Cycle until Bof is True }
  begin
    { Process each record here }
    ⋮
```

```
      CustTable.Prior; { Bof False on success; Bof True when Prior fails on first record }
    end;
  finally
    CustTable.EnableControls; { Display new current row in controls }
  end;
```

Marking and returning to records

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. *TDataSet* introduces a bookmarking feature that consists of a *Bookmark* property and five bookmark methods.

TDataSet implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. *TDataSet* descendants vary in the level of support they provide for bookmarks. None of the dbExpress datasets add any support for bookmarks. ADO datasets can support bookmarks, depending on the underlying database tables. BDE datasets, InterBase express datasets, and client datasets always support bookmarks.

The Bookmark property

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

The GetBookmark method

To create a bookmark, you must declare a variable of type *TBookmark* in your application, then call *GetBookmark* to allocate storage for the variable and set its value to a particular location in a dataset. The *TBookmark* type is a Pointer.

The GotoBookmark and BookmarkValid methods

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. Before calling *GotoBookmark*, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record.

The CompareBookmarks method

You can also call *CompareBookmarks* to see if a bookmark you want to move to is different from another (or the current) bookmark. If the two bookmarks refer to the same record (or if both are **nil**), *CompareBookmarks* returns 0.

The FreeBookmark method

FreeBookmark frees the memory allocated for a specified bookmark when you no longer need it. You should also call *FreeBookmark* before reusing an existing bookmark.

A bookmarking example

The following code illustrates one use of bookmarking:

```
procedure DoSomething (const Tbl: TTable)
var
  Bookmark: TBookmark;
begin
  Bookmark := Tbl.GetBookmark; { Allocate memory and assign a value }
  Tbl.DisableControls; { Turn off display of records in data controls }
  try
    Tbl.First; { Go to first record in table }
    while not Tbl.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      ⋮
      Tbl.Next;
    end;
  finally
    Tbl.GotoBookmark(Bookmark);
    Tbl.EnableControls; { Turn on display of records in data controls, if necessary }
    Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

Searching datasets

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods. These methods enable you to search on any type of columns in any dataset.

Note Some *TDataSet* descendants introduce an additional family of methods for searching based on an index. For information about these additional methods, see “Using Indexes to search for records” on page 18-27.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is “Professional Divers, Ltd.”:

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
```

```
    SearchOptions := [loPartialKey];
    LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
  end;
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *True* if it finds a matching record, *False* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is “Professional Divers, Ltd.”, and returns the company name, a contact person, and a phone number for the company:

```
var
  LookupResults: Variant;
begin
  LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
     'Company;Contact; Phone');
end;
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant. For more information about Variant arrays, see the online help.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
    'Company; Addr1; Addr2; State; Zip');
end;
```

Like *Locate*, *Lookup* uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

Displaying and editing a subset of data using filters

An application is frequently interested in only a subset of records from a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. In each case, you can use filters to restrict an application's access to a subset of all records in the dataset.

With unidirectional datasets, you can only limit the records in the dataset by using a query that restricts the records in the dataset. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's *Filter* property or coded into its *OnFilterRecord* event handler. Filter conditions are based on the values in any specified number of fields in a dataset, regardless of whether those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

Note Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

Enabling and disabling filtering

Enabling filters on a dataset is a three-step process:

1. Create a filter.
2. Set filter options for string-based filter tests, if necessary.
3. Set the *Filtered* property to *True*.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to *False*.

Creating filters

There are two ways to create a filter for a dataset:

- Specify simple filter conditions in the *Filter* property. *Filter* is especially useful for creating and applying filters at runtime.
- Write an *OnFilterRecord* event handler for simple or complex filter conditions. With *OnFilterRecord*, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

Setting the Filter property

To create a filter using the *Filter* property, set the value of the property to a string that contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

You can also supply a value for *Filter* based on text supplied by the user. For example, the following statement assigns the text in from edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
```

You can, of course, create a string based on both hard-coded text and user-supplied data:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

Blank field values do not appear unless they are explicitly included in the filter:

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

Note After you specify a value for *Filter*, to apply the filter to the dataset, set the *Filtered* property to *True*.

Filters can compare field values to literals and to constants using the following comparison and logical operators:

Table 18.4 Comparison and logical operators that can appear in a filter

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both *True*
NOT	Tests that the following statement is not *True*
OR	Tests that at least one of two statements is *True*
+	Adds numbers, concatenates strings, ads numbers to date/time values (only available for some drivers)
-	Subtracts numbers, subtracts dates, or subtracts a number from a date (only available for some drivers)
*	Multiplies two numbers (only available for some drivers)
/	Divides two numbers (only available for some drivers)
*	wildcard for partial comparisons (*FilterOptions* must include *foPartialCompare*)

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

Note When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

Writing an OnFilterRecord event handler

You can write code to filter records using the *OnFilterRecord* events generated by the dataset for each record it retrieves. This event handler implements a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your *OnFilterRecord* handler sets its *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with the State field set to “CA”:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter’s conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly-coded as possible to avoid adversely affecting the performance.

Switching filter event handlers at runtime

You can code any number of *OnFilterRecord* event handlers and switch among them at runtime. For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

Setting filter options

The *FilterOptions* property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

Table 18.5 FilterOptions values

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.
foNoPartialCompare	Disable partial string matching; that is, don’t match strings that end with an asterisk (*).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

Navigating records in a filtered dataset

There are four dataset methods that navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

Table 18.6 Filtered dataset navigational methods

Method	Purpose
FindFirst	Move to the first record that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
FindLast	Move to the last record that matches the current filter criteria.
FindNext	Moves from the current record in the filtered dataset to the next one.
FindPrior	Move from the current record in the filtered dataset to the previous one.

For example, the following statement finds the first filtered record in a dataset:

```
DataSet1.FindFirst;
```

Provided that you set the *Filter* property or create an *OnFilterRecord* event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

Note If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return *False*. You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset and you call *FindNext*, the method returns *False*, and the current record is unchanged.

Modifying data

You can use the following dataset methods to insert, update, and delete data if the read-only *CanModify* property is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor intervenes. (Intervening factors include the *ReadOnly* property on some datasets or the *RequestLive* property on *TQuery* components.)

Table 18.7 Dataset methods for inserting, updating, and deleting data

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

Editing records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*.

When a dataset transitions to *dsEdit* mode, it first receives a *BeforeEdit* event. After the transition to edit mode is successfully completed, the dataset receives an *AfterEdit* event. Typically, these events are used for updating the user interface to indicate the current state of the dataset. If the dataset can't be put into edit mode for some reason, an *OnEditError* event occurs, where you can inform the user of the problem or try to correct the situation that prevented the dataset from entering edit mode.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Note Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you have a navigator component on your form, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, *Edit* and *Post* are often used together. For example,

```
with CustTable do
begin
  Edit;
```

```
    FieldValues['CustNo'] := 1234;
    Post;
  end;
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record. If you are not caching updates, posting writes the change back to the database. If you are caching updates, the change is written to a temporary buffer, where it stays until the dataset's *ApplyUpdates* method is called.

Adding new records

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*.

When a dataset transitions to *dsInsert* mode, it first receives a *BeforeInsert* event. After the transition to insert mode is successfully completed, the dataset receives first an *OnNewRecord* event hand then an *AfterInsert* event. You can use these events, for example, to provide initial values to newly inserted records:

```
procedure TForm1.OrdersTableNewRecord(DataSet: TDataSet);
begin
  DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;
```

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

Note Even if a dataset is in *dsInsert* state, adding records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

Post writes the new record to the database, or, if you are caching updates, *Post* writes the record to an in-memory cache. To write cached inserts and appends to the database, call the dataset's *ApplyUpdates* method.

Inserting records

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when using cached updates), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at its current position.
- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

Appending records

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when using cached updates), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

Deleting records

Use the *Delete* method to delete the current record in an active dataset. When the *Delete* method is called,

- The dataset receives a *BeforeDelete* event.
- The dataset attempts to delete the current record.
- The dataset returns to the *dsBrowse* state.
- The dataset receives an *AfterDelete* event.

If want to prevent the deletion in the *BeforeDelete* event handler, you can call the global *Abort* procedure:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

If *Delete* fails, it generates an *OnDeleteError* event. If the *OnDeleteError* event handler can't correct the problem, the dataset remains in *dsEdit* state. If *Delete* succeeds, the dataset reverts to the *dsBrowse* state and the record that followed the deleted record becomes the current record.

If you are caching updates, the deleted record is not removed from the underlying database table until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

Posting data

After you finish editing a record, you must call the *Post* method to write out your changes. The *Post* method behaves differently, depending on the dataset's state and on whether you are caching updates.

- If you are not caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to the database and returns the dataset to the *dsBrowse* state.
- If you are caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to an internal cache and returns the dataset to the *dsBrowse* state. The edits are net written to the database until you call *ApplyUpdates*.
- If the dataset is in the *dsSetKey* state, *Post* returns the dataset to the *dsBrowse* state.

Regardless of the initial state of the dataset, *Post* generates *BeforePost* and *AfterPost* events, before and after writing the current changes. You can use these events to update the user interface, or prevent the dataset from posting changes by calling the *Abort* procedure. If the call to *Post* fails, the dataset receives an *OnPostError* event, where you can inform the user of the problem or attempt to correct it.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The *Append* and *Insert* methods also implicitly post any pending data.

Warning The *Close* method does not call *Post* implicitly. Use the *BeforeClose* event to post any pending edits explicitly.

Canceling changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the *Cancel* method for the dataset. A call to *Cancel* always returns a dataset to *dsBrowse* state.

If the dataset was in *dsEdit* or *dsInsert* mode when your application called *Cancel*, it receives *BeforeCancel* and *AfterCancel* events before and after the current record is restored to its original values.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

Modifying entire records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Table 18.8 Methods that work with entire records

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit *Post*.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. The application must perform an explicit *Post*.

These method take an array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, *AppendRecord* adds a record to the end of the dataset and *InsertRecord* inserts a record after the current cursor position. For indexed datasets, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a *Post*.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, SetFields assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a *TTable* component called *CountryTable* were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

This code assigns values to the Area and Population fields and then posts them to the database. The three NULL pointers act as place holders for the first three columns to preserve their current contents.

Calculating fields

Using the Fields editor, you can define calculated fields for your datasets. When a dataset contains calculated fields, you provide the code to calculate those field's values in an *OnCalcFields* event handler. For details on how to define calculated fields using the Fields editor, see "Defining a calculated field" on page 19-7.

The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when

- A dataset is opened.
- The dataset enters edit mode.
- A record is retrieved from the database.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a record are edited (the fourth condition above).

Caution *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or a linked dataset if it is part of a master-detail relationship), because this leads to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, causing another *Post*, and so on.

When *OnCalcFields* executes, a dataset enters *dsCalcFields* mode. This state prevents modifications or additions to the records except for the calculated fields the handler is designed to modify. The reason for preventing other modifications is because *OnCalcFields* uses the values in other fields to derive calculated field values. Changes to those other fields might otherwise invalidate the values assigned to calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Types of datasets

"Using TDataSet descendants" on page 18-2 classifies *TDataSet* descendants by the method they use to access their data. Another useful way to classify *TDataSet* descendants is to consider the type of server data they represent. Viewed this way, there are three basic classes of datasets:

- **Table-type datasets**: Table-type datasets represent a single table from the database server, including all of its rows and columns. Table-type datasets include *TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*.

  Table-type datasets let you take advantage of indexes defined on the server. Because there is a one-to-one correspondence between database table and dataset, you can use server indexes that are defined for the database table. Indexes allow your application to sort the records in the table, speed searches and lookups, and can form the basis of a master/detail relationship. Some table-type datasets also take advantage of the one-to-one relationship between dataset and database table to let you perform table-level operations such as creating and deleting database tables.

- **Query-type datasets**: Query-type datasets represent a single SQL command, or query. Queries can represent the result set from executing a command (typically a SELECT statement), or they can execute a command that does not return any records (for example, an UPDATE statement). Query-type datasets include *TQuery*, *TADOQuery*, *TSQLQuery*, and *TIBQuery*.

  To use a query-type dataset effectively, you must be familiar with SQL and your server's SQL implementation, including limitations and extensions to the SQL-92 standard. If you are new to SQL, you may want to purchase a third party book that covers SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

- **Stored procedure-type datasets**: Stored procedure-type datasets represent a stored procedure on the database server. Stored procedure-type datasets include *TStoredProc*, *TADOStoredProc*, *TSQLStoredProc*, and *TIBStoredProc*.

  A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. They typically handle frequently-repeated database-related tasks, and are especially useful for operations that act on large numbers of records or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

  - Taking advantage of the server's usually greater processing power and speed.
  - Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a SELECT query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

Note You can usually use a query-type dataset to execute stored procedures because most servers provide extensions to SQL for working with stored procedures. Each server, however, uses its own syntax for this. If you choose to use a query-type dataset instead of a stored procedure-type dataset, see your server documentation for the necessary syntax.

In addition to the datasets that fall neatly into these three categories, *TDataSet* has some descendants that fit into more than one category:

- *TADODataSet* and *TSQLDataSet* have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are most similar to query-type datasets, although *TADODataSet* lets you specify an index like a table-type dataset.
- *TClientDataSet* represents the data from another dataset. As such, it can represent a table, query, or stored procedure. *TClientDataSet* behaves most like a table-type dataset, because of its index support. However, it also has some of the features of queries and stored procedures: the management of parameters and the ability to execute without retrieving a result set.
- Some other client datasets (*TBDEClientDataSet* and *TSQLClientDataSet*) have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are like *TClientDataSet*, including parameter support, indexes, and the ability to execute without retrieving a result set.
- *TIBDataSet* can represent both queries and stored procedures. In fact, it can represent multiple queries and stored procedures simultaneously, with separate properties for each.

Using table-type datasets

To use a table-type dataset,

1. Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that contains the table you want to use. Each table-type dataset does this differently, but typically you specify a database connection component:
    - For *TTable*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.

- For *TADOTable*, specify a *TADOConnection* component using the *Connection* property.
- For *TSQLTable*, specify a *TSQLConnection* component using the *SQLConnection* property.
- For *TIBTable*, specify a *TIBConnection* component using the *Database* property.

For information about using database connection components, see Chapter 17, "Connecting to databases".

3 Set the *TableName* property to the name of the table in the database. You can select tables from a drop-down list if you have already identified a database connection component.

4 Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the dataset. The data source component is used to pass a result set from the dataset to data-aware components for display.

Advantages of using table-type datasets

The main advantage of using table-type datasets is the availability of indexes. Indexes enable your application to

- Sort the records in the dataset.
- Locate records quickly.
- Limit the records that are visible.
- Establish master/detail relationships.

In addition, the one-to-one relationship between table-type datasets and database tables enables many of them to be used for

- Controlling Read/write access to tables
- Creating and deleting tables
- Emptying tables
- Synchronizing tables

Sorting records with indexes

An index determines the display order of records in a table. Typically, records appear in ascending order based on a primary, or default, index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.
- A list of columns on which to sort (not available on servers that aren't SQL-based).

Indexes let you present the data from a table in different orders. On SQL-based tables, this sort order is implemented by using the index to generate an ORDER BY clause in a query that fetches the table's records. On other tables (such as Paradox and dBASE tables), the index is used by the data access mechanism to present records in the desired order.

Obtaining information about indexes

You application can obtain information about server-defined indexes from all table-type datasets. To obtain a list of available indexes for the dataset, call the *GetIndexNames* method. *GetIndexNames* fills a string list with valid index names. For example, the following code fills a listbox with the names of all indexes defined for the *CustomersTable* dataset:

```
CustomersTable.GetIndexNames(ListBox1.Items);
```

Note For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. You can still change the index back to a primary index on a Paradox table after using an alternative index, however, by setting the *IndexName* property to a blank string.

To obtain information about the fields of the current index, use the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list the field components for the columns that comprise the index.

The following code illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
```

Note *IndexFieldCount* is not valid for a dBASE table opened on an expression index.

Specifying an index with IndexName

Use the *IndexName* property to cause an index to be active. Once active, an index determines the order of records in the dataset. (It can also be used as the basis for a master-detail link, an index-based search, or index-based filtering.)

To activate an index, set the *IndexName* property to the name of the index. In some database systems, primary indexes do not have names. To activate one of these indexes, set *IndexName* to a blank string.

At design-time, you can select an index from a list of available indexes by clicking the property's ellipsis button in the Object Inspector. At runtime set *IndexName* using a ***String*** literal or variable. You can obtain a list of available indexes by calling the *GetIndexNames* method.

The following code sets the index for *CustomersTable* to *CustDescending*:

```
CustomersTable.IndexName := 'CustDescending';
```

Creating an index with IndexFieldNames

If there is no defined index that implements the sort order you want, you can create a pseudo-index using the *IndexFieldNames* property.

Note *IndexName* and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other.

The value of *IndexFieldNames* is a string. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for *PhoneTable* based on *LastName*, then *FirstName*:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

Note If you use *IndexFieldNames* on Paradox and dBASE tables, the dataset attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

Using Indexes to search for records

You can search against any dataset using the *Locate* and *Lookup* methods of *TDataSet*. However, by explicitly using indexes, some table-type datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods.

ADO datasets all support the *Seek* method, which moves to a record based on a set of field values for fields in the current index. *Seek* lets you specify where to move the cursor relative to the first or last matching record.

TTable and all types of client dataset support similar indexed-based searches, but use a combination of related methods. The following table summarizes the six related methods provided by *TTable* and client datasets to support index-based searches:

Table 18.9 Index-based search methods

Method	Purpose
EditKey	Preserves the current contents of the search key buffer and puts the dataset into *dsSetKey* state so your application can modify existing search criteria prior to executing a search.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single method.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single method.
GotoKey	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
GotoNearest	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
SetKey	Clears the search key buffer and puts the table into *dsSetKey* state so your application can specify new search criteria prior to executing a search.

GotoKey and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

GotoNearest and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

Executing a search with Goto methods

To execute a search using *Goto* methods, follow these general steps:

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset (see “Sorting records with indexes” on page 18-25). To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Put the dataset in *dsSetKey* state by calling the *SetKey* method.
4. Specify the value(s) to search on in the *Fields* property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button’s *OnClick* event, uses the *GotoKey* method to move to the first record where the first field in the index has a value that exactly matches the text in an edit box:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
```

If a record exists with “Sm” as the first two characters of the first indexed field’s value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns *False*.

Executing a search with Find methods

The *Find* methods do the same thing as the *Goto* methods, except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search. To execute a search using *Find* methods, follow these general steps:

1 Specify the index to use for the search. This is the same index that sorts the records in the dataset (see "Sorting records with indexes" on page 18-25). To specify the index, use the *IndexName* or *IndexFieldNames* property.

2 Open the dataset.

3 Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

Note *FindNearest* can only be used for string fields.

Specifying the current record after a successful search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

Searching on partial keys

If the dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the dataset's current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For table-type datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

Repeating or extending a search

Each time you call *SetKey* or *FindKey*, the method clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*.

For example, suppose you have already executed a search of the Employee table based on the City field of the "CityIndex" index. Suppose further that "CityIndex" includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

```
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
```

Limiting records with ranges

You can temporarily view and edit a subset of data for any dataset by using filters (see “Displaying and editing a subset of data using filters” on page 18-12). Some table-type datasets support an additional way to access a subset of available records, called ranges.

Ranges only apply to *TTable* and to client datasets. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges.

Understanding the differences between ranges and filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than “Jones” and less than “Smith”. Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to sort records, you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query-type dataset to select data. For details on specifying a query, see “Using query-type datasets” on page 18-41.

Specifying Ranges

There are two mutually exclusive ways to specify a range:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.
- Specify both endpoints at once using *SetRange*.

Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the *Fields* property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a *TSQLClientDataSet* component named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

Tip To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 18-33.

Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

Warning Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName* method. For example,

```
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range. For information about applying and canceling ranges, see "Applying or canceling a range" on page 18-33.

Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

SetRange takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statement establishes a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith." The value specification could also be:

```
Contacts['LastName'] := 'Sm';
```

This statement includes records that have *LastName* greater than or equal to "Sm."

Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a dataset to *True* to exclude records equal to ending range. For example,

```
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

Modifying a range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

The process for editing and applying a range involves these general steps:

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the *Fields* property overwrite the current index values to use when applying the range.

Tip If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range. For more information about ranges based on partial keys, see “Specifying a range based on partial keys” on page 18-32.

Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range.

Applying or canceling a range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* method. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```
⋮
MyTable.CancelRange;
⋮
{later on, use the same range again. No need to call SetRangeStart, etc.}
MyTable.ApplyRange;
⋮
```

Creating master/detail relationships

Table-type datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Table-type datasets support master/detail relationships in two very distinct ways:

- All table-type datasets can act as the detail of another dataset by linking cursors. This process is described in "Making the table a detail of another dataset" below.
- *TTable*, *TSQLTable*, and all client datasets can act as the master in a master/detail relationship that uses nested detail tables. This process is described in "Using nested detail tables" on page 18-36.

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is any type of dataset. With nested details, the type of dataset that can act as the detail table is limited, but they provide for more options in how to display the data. If the master is a client dataset, nested details provide a more robust mechanism for applying cached updates.

Making the table a detail of another dataset

A table-type dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the table gets data from the master table. This data source can be linked to any type of dataset. For instance, by specifying a query's data source in this property, you can link a client dataset as the detail of the query, so that the client dataset tracks events occurring in the query.

The dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the detail dataset, first specify the index in the detail dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you specify the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the index fields in the detail table. To link datasets on multiple column names, separate field names with semicolons:

```
Parts.MasterFields := 'OrderNo;ItemNo';
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the *CustomersTable* table, and the detail table is *OrdersTable*. The example uses the BDE-based *TTable* component, but you can use the same methods to link any table-type datasets.

1 Place two *TTable* components and two *TDataSource* components in a data module.

2 Set the properties of the first *TTable* component as follows:

- *DatabaseName*: DBDEMOS
- *TableName*: CUSTOMER
- *Name*: CustomersTable

3 Set the properties of the second *TTable* component as follows:

- *DatabaseName*: DBDEMOS
- *TableName*: ORDERS
- *Name*: OrdersTable

4 Set the properties of the first *TDataSource* component as follows:

- *Name*: CustSource
- *DataSet*: CustomersTable

5 Set the properties of the second *TDataSource* component as follows:

- *Name*: OrdersSource
- *DataSet*: OrdersTable

6 Place two *TDBGrid* components on a form.

7 Choose File|Use Unit to specify that the form should use the data module.

8 Set the *DataSource* property of the first grid component to "CustSource", and set the *DataSource* property of the second grid to "OrdersSource".

9 Set the *MasterSource* property of *OrdersTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).

10 Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:

- In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
- Select *CustNo* in both the Detail Fields and Master Fields field lists.
- Click the Add button to add this join condition. In the Joined Fields list, "CustNo -> CustNo" appears.
- Choose OK to commit your selections and exit the Field Link Designer.

11 Set the *Active* properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.

12 Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

Using nested detail tables

A nested table is a detail dataset that is the value of a single dataset field in another (master) dataset. For datasets that represent server data, a nested detail dataset can only be used for a dataset field on the server. *TClientDataSet* components do not represent server data, but they can also contain dataset fields if you create a dataset for them that contains nested details, or if they receive data from a provider that is linked to the master table of a master/detail relationship.

Note For *TClientDataSet*, using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server.

To use nested detail sets, the *ObjectView* property of the master dataset must be *True*. When your table-type dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on how this works, see "Displaying dataset fields" on page 19-26.

Alternately, you can display and edit detail datasets in data-aware controls by using a separate dataset component for the detail set. At design time, create persistent fields for the fields in your (master) dataset, using the Fields Editor: right click the master dataset and choose Fields Editor. Add a new persistent field to your dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of the detail table. You must also add persistent fields for any other fields used in your master dataset.

The dataset component for the detail table is a dataset descendant of a type allowed by the master table. *TTable* components only allow *TNestedDataSet* components as nested datasets. *TSQLTable* components allow other *TSQLTable* components. *TClientDataset* components allow other client datasets. Choose a dataset of the appropriate type from the Component palette and add it to your form or data module. Set this detail dataset's *DataSetField* property to the persistent DataSet field in the master dataset. Finally, place a data source component on the form or data module and set its *DataSet* property to the detail dataset. Data-aware controls can use this data source to access the data in the detail set.

Controlling Read/write access to tables

By default when a table-type dataset is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

Note This is not true for *TClientDataSet*, which determines whether users can edit data from information that the dataset provider supplies with data packets. It is also not true for *TSQLTable*, which is a unidirectional dataset, and hence always read-only.

When the table opens, you can check the *CanModify* property to ascertain whether the underlying database (or the dataset provider) allows users to edit the data in the table. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided the table's *ReadOnly* property is *False*.

ReadOnly determines whether a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening the table.

Note *ReadOnly* is implemented on all table-type datasets except *TSQLTable*, which is always read-only.

Creating and deleting tables

Some table-type datasets let you create and delete the underlying tables at design time or at runtime. Typically, database tables are created and deleted by a database administrator. However, it can be handy during application development and testing to create and destroy database tables that your application can use.

Creating tables

TTable and *TIBTable* both let you create the underlying database table without using SQL. Similarly, *TClientDataSet* lets you create a dataset when you are not working with a dataset provider. Using *TTable* and *TClientDataSet*, you can create the table at design time or runtime. *TIBTable* only lets you create tables at runtime.

Before you can create the table, you must be set properties to specify the structure of the table you are creating. In particular, you must specify

- The database that will host the new table. For *TTable*, you specify the database using the *DatabaseName* property. For *TIBTable*, you must use a *TIBDataBase* component, which is assigned to the *Database* property. (Client datasets do not use a database.)
- The type of database (*TTable* only). Set the *TableType* property to the desired type of table. For Paradox, dBASE, or ASCII tables, set *TableType* to *ttParadox*, *ttDBase*, or *ttASCII*, respectively. For all other table types, set *TableType* to *ttDefault*.

- The name of the table you want to create. Both *TTable* and *TIBTable* have a *TableName* property for the name of the new table. Client datasets do not use a table name, but you should specify the *FileName* property before you save the new table. If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered. To avoid overwriting an existing table, you can check the *Exists* property at runtime. *Exists* is only available on *TTable* and *TIBTable*.
- The fields for the new table. There are two ways to do this:
  - You can add field definitions to the *FieldDefs* property. At design time, double-click the *FieldDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the *AddFieldDef* method to add each new field definition. For each new field definition, set the properties of the *TFieldDef* object to specify the desired attributes of the field.
  - You can use persistent field components instead. At design time, double-click on the dataset to bring up the *Fields* editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.
- Indexes for the new table (optional). At design time, double-click the *IndexDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of index definitions. At runtime, clear any existing index definitions, and then use the *AddIndexDef* method to add each new index definition. For each new index definition, set the properties of the *TIndexDef* object to specify the desired attributes of the index.

Note You can't define indexes for the new table if you are using persistent field components instead of field definition objects.

To create the table at design time, right-click the dataset and choose Create Table (*TTable*) or Create Data Set (*TClientDataSet*). This command does not appear on the context menu until you have specified all the necessary information.

To create the table at runtime, call the *CreateTable* method (*TTable* and *TIBTable*) or the *CreateDataSet* method (*TClientDataSet*).

Note You can set up the definitions at design time and then call the *CreateTable* (or *CreateDataSet*) method at runtime to create the table. However, to do so you must indicate that the definitions specified at runtime should be saved with the dataset component. (by default, field and index definitions are generated dynamically at runtime). Specify that the definitions should be saved with the dataset by setting its *StoreDefs* property to *True*.

Tip If you are using *TTable*, you can preload the field definitions and index definitions of an existing table at design time. Set the *DatabaseName* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *DatabaseName* and *TableName* to specify the table you want to create, canceling any prompts to rename the existing table.

Note When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, and dataset fields).

The following code creates a new table at runtime and associates it with the DBDEMOS alias. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```
var
  TableFound: Boolean;
begin
  with TTable.Create(nil) do // create a temporary TTable component
  begin
    try
      { set properties of the temporary TTable component }
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      { define fields for the new table }
      FieldDefs.Clear;
      with FieldDefs.AddFieldDef do begin
        Name := 'First';
        DataType := ftString;
        Size := 20;
        Required := False;
      end;
      with FieldDefs.AddFieldDef do begin
        Name := 'Second';
        DataType := ftString;
        Size := 30;
        Required := False;
      end;
      { define indexes for the new table }
      IndexDefs.Clear;
      with IndexDefs.AddIndexDef do begin
        Name := '';
        Fields := 'First';
        Options := [ixPrimary];
      end;
      TableFound := Exists; // check whether the table already exists
      if TableFound then
        if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
             mtConfirmation, mbYesNoCancel, 0) = mrYes then
          TableFound := False;
      if not TableFound then
        CreateTable; // create the table
    finally
      Free; // destroy the temporary TTable when done
    end;
  end;
end;
```

Deleting tables

TTable and *TIBTable* let you delete tables from the underlying database table without using SQL. To delete a table at runtime, call the dataset's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
```

Caution When you delete a table with *DeleteTable*, the table and all its data are gone forever.

If you are using *TTable*, you can also delete tables at design time: Right-click the table component and select Delete Table from the context menu. The Delete Table menu pick is only present if the table component represents an existing database table (the *DatabaseName* and *TableName* properties specify an existing table).

Emptying tables

Many table-type datasets supply a single method that lets you delete all rows of data in the table.

- For *TTable* and *TIBTable*, you can delete all the records by calling the *EmptyTable* method at runtime:

```
PhoneTable.EmptyTable;
```

- For *TADOTable*, you can use the *DeleteRecords* method.

```
PhoneTable.DeleteRecords;
```

- For *TSQLTable*, you can use the *DeleteRecords* method as well. Note, however, that the *TSQLTable* version of *DeleteRecords* never takes any parameters.

```
PhoneTable.DeleteRecords;
```

- For client datasets, you can use the *EmptyDataSet* method.

```
PhoneTable.EmptyDataSet;
```

Note For tables on SQL servers, these methods only succeed if you have DELETE privilege for the table.

Caution When you empty a dataset, the data you delete is gone forever.

Synchronizing tables

If you have two or more datasets that represent the same database table but do not share a data source component, then each dataset has its own view on the data and its own current record. As users access records through each datasets, the components' current records will differ.

If the datasets are all instances of *TTable*, or all instances of *TIBTable*, or all client datasets, you can force the current record for each of these datasets to be the same by calling the *GotoCurrent* method. *GotoCurrent* sets its own dataset's current record to the current record of a matching dataset. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

Tip If your application needs to synchronize datasets in this manner, put the datasets in a data module and add the unit for the data module to the uses clause of each unit that accesses the tables.

To synchronize datasets from separate forms, you must add one form's unit to the uses clause of the other, and you must qualify at least one of the dataset names with its form name. For example:

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

Using query-type datasets

To use a query-type dataset,

1. Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server to query. Each query-type dataset does this differently, but typically you specify a database connection component:
   - For *TQuery*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
   - For *TADOQuery*, specify a *TADOConnection* component using the *Connection* property.
   - For *TSQLQuery*, specify a *TSQLConnection* component using the *SQLConnection* property.
   - For *TIBQuery*, specify a *TIBConnection* component using the *Database* property.

   For information about using database connection components, see Chapter 17, "Connecting to databases".
3. Specify an SQL statement in the *SQL* property of the dataset, and optionally specify any parameters for the statement. For more information, see "Specifying the query" on page 18-42 and "Using parameters in queries" on page 18-43.
4. If the query data is to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the query-type dataset. The data source component forwards the results of the query (called a *result set*) to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. To execute queries that only perform an action on a table and return no result set, use the *ExecSQL* method at runtime. If you plan to execute the query more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the query. For information about preparing a query, see "Preparing queries" on page 18-47.

Specifying the query

For true query-type datasets, you use the *SQL* property to specify the SQL statement for the dataset to execute. Some datasets, such as *TADODataSet*, *TSQLDataSet*, and client datasets, use a *CommandText* property to accomplish the same thing.

Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

Queries that do not return records include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution. In most cases, the SQL command must be only one complete SQL statement, although that statement can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). Some servers also support “batch” syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements when you specify the query.

The SQL statements used by queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user’s view of and access to data on the fly at runtime without having to alter the SQL statement. For more information about parameterized queries, see “Using parameters in queries” on page 18-43.

Specifying a query using the SQL property

When using a true query-type dataset *(TQuery*, *TADOQuery*, *TSQLQuery*, or *TIBQuery)*, assign the query to the *SQL* property. The *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to modify and debug the query if you divide the statement into logical units:

```
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT CustNo, OrderNO, SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER BY SaleDate');
MyQuery.Open;
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the ORDER BY clause already exists on the third line of the statement. It is referenced via the *SQL* property using an index of 2.

```
MyQuery.SQL[2] := ‘ORDER BY OrderNo’;
```

Note The dataset must be closed when you specify or modify the SQL property.

At design time, use the String List editor to specify the query. Click the ellipsis button by the *SQL* property in the Object Inspector to display the String List editor.

Note With some versions of Delphi, if you are using *TQuery*, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select the query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use SQL Builder, open it and use its online help.

Because the *SQL* property is a *TStrings* object, you can load the text of the query from a file by calling the *TStrings.LoadFromFile* method:

```
MyQuery.SQL.LoadFromFile('custquery.sql');
```

You can also use the *Assign* method of the *SQL* property to copy the contents of a string list object into the *SQL* property. The *Assign* method automatically clears the current contents of the *SQL* property before copying the new statement:

```
MyQuery.SQL.Assign(Memo1.Lines);
```

Specifying a query using the CommandText property

When using *TADODataSet*, *TSQLDataSet*, or a client dataset, assign the text of the query statement to the *CommandText* property:

```
MyQuery.CommandText := 'SELECT CustName, Address FROM Customer';
```

At design time, you can type the query directly into the Object Inspector, or, if the dataset already has an active connection to the database, you can click the elipsis button by the *CommandText* property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

Using parameters in queries

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, *:Name*, *:Capital*, and *:Population* are placeholders for actual values supplied to the statement at runtime by your application. Note that the names of parameters begin with a colon. The colon is required so that the parameter names can be distinguished from literal values. You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. *TQuery*, *TIBQuery*, *TSQLQuery*, and client datasets use the *Params* property to store these values. *TADOQuery* uses the *Parameters* property instead. *Params* (or *Parameters*) is a collection of parameter objects (*TParam* or *TParameter*), where each object represents a single parameter. When you specify the text for the query, the dataset generates this set of parameter objects, and (depending on the dataset type) initializes any of their properties that it can deduce from the query.

Note You can suppress the automatic generation of parameter objects in response to changing the query text by setting the *ParamCheck* property to *False*. This is useful for data definition language (DDL) statements that contain parameters as part of the DDL statement that are not parameters for the query itself. For example, the DDL statement to create a stored procedure may define parameters that are part of the stored procedure. By setting *ParamCheck* to *False*, you prevent these parameters from being mistaken for parameters of the query.

Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

Tip It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is “Number,” then its corresponding parameter would be “:Number”. Using matching names is especially important if the dataset uses a datasource to obtain parameter values from another dataset. This process is described in “Establishing master/detail relationships using parameters” on page 18-46.

Supplying parameters at design time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the Object Inspector. If the SQL statement does not contain any parameters, no objects are listed in the collection editor.

Note The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for query parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the Object Inspector to modify its properties.

When using the *Params* property (*TParam* objects), you will want to inspect or modify the following:

- The *DataType* property lists the data type for the parameter’s value. For some datasets, this value may be correctly initialized. If the dataset did not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.

The *DataType* property lists the logical data type for the parameter. In general, these data types conform to server data types. For specific logical type-to-server data type mappings, see the documentation for the data access mechanism (BDE, dbExpress, InterBase).

- The *ParamType* property lists the type of the selected parameter. For queries, this is always *ptInput*, because queries can only contain input parameters. If the value of *ParamType* is *ptUnknown*, change it to *ptInput*.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

When using the *Parameters* property (*TParameter* objects), you will want to inspect or modify the following:

- The *DataType* property lists the data type for the parameter's value. For some data types, you must provide additional information:
    - The *NumericScale* property indicates the number of decimal places for numeric parameters.
    - The *Precision* property indicates the total number of digits for numeric parameters.
    - The *Size* property indicates the number of characters in string parameters.
- The *Direction* property lists the type of the selected parameter. For queries, this is always *pdInput*, because queries can only contain input parameters.
- The *Attributes* property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

Supplying parameters at runtime

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name (not available for *TADOQuery*)
- *Params* or *Parameters* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* or *Parameters.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:

```
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the :Capital parameter is the first parameter in the SQL statement):

```
SQLQuery1.Params[0].AsString := Edit1.Text;
```

The command line below sets three parameters at once, using the *Params.ParamValues* property:

```
Query1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Note that *ParamValues* uses Variants, avoiding the need to cast values.

Establishing master/detail relationships using parameters

To set up a master/detail relationship where the detail set is a query-type dataset, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, all query-type datasets except *TIBQuery* provide an easier mechanism using the *DataSource* property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, query-type datasets attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

The first step is to set up the Customer dataset:

1 Add a table-type dataset to your application and bind it to the Customer table.

2 Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.

3 Add a query-type dataset and set its *SQL* property to

   ```
   SELECT CustID, OrderNo, SaleDate
   FROM Orders
   WHERE CustID = :ID
   ```

   Note that the name of the parameter is the same as the name of the field in the master (Customer) table.

4 Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail dataset is not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by *CustomersSource*. *CustomersSource* gets its data from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called “ID,” the value from the *ID* field in the current record of the master dataset is assigned to the *:ID* parameter for the detail dataset’s SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset’s SELECT statement executes to retrieve all orders based on the current customer id.

Preparing queries

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the data access layer and the database server for parsing, resource allocation, and optimization. In some datasets, the dataset may perform additional setup operations when preparing the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by setting the *Prepared* property to *True*. If you do not prepare a query before executing it, the dataset automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though the dataset prepares queries for you, you can improve performance by explicitly preparing the dataset before you open it the first time.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you add a parameter).

Note When you change the text of the *SQL* property for a query, the dataset automatically closes and unprepares the query.

Executing queries that don’t return a result set

When a query returns a set of records (such as a SELECT query), you execute the query the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often SQL commands do not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records).

For all query-type datasets, you can execute a query that does not return a result set by calling *ExecSQL*:

```
CustomerQuery.ExecSQL;  { query does not return a result set }
```

Tip If you are executing the query multiple times, it is a good idea to set the *Prepared* property to *True*.

Although the query does not return any records, you may want to know the number of records it affected (for example, the number of records deleted by a DELETE query). The *RowsAffected* property gives the number of affected records after a call to *ExecSQL*.

Tip When you do not know at design time whether the query returns a result set (for example, if the user supplies the query dynamically at runtime), you can code both types of query execution statements in a **try...except** block. Put a call to the *Open* method in the **try** clause. An action query is executed when the query is activated with the *Open* method, but an exception occurs in addition to that. Check the exception, and suppress it if it merely indicates the lack of a result set. (For example, *TQuery* indicates this by an *ENoResultSet* exception.)

Using unidirectional result sets

When a query-type dataset returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source. Unless you are using dbExpress, this cursor is bi-directional by default. A bi-directional cursor can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries.

If you do not need to be able to navigate backward through a result set, *TQuery* and *TIBQuery* let you improve query performance by requesting a unidirectional cursor instead. To request a unidirectional cursor, set the *UniDirectional* property to *True*.

Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
if not (CustomerQuery.Prepared) then
begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepared := True;
end;
CustomerQuery.Open;  { returns a result set with a one-way cursor }
```

Note Do not confuse the *UniDirectional* property with a unidirectional dataset. Unidirectional datasets (*TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*) use dbExpress, which only returns unidirectional cursors. In addition to restricting the ability to navigate backwards, unidirectional datasets do not buffer records, and so have additional limitations (such as the inability to use filters).

Using stored procedure-type datasets

How your application uses a stored procedure depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

In general terms, to access a stored procedure on a server, an application must:

1. Place the appropriate dataset component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that defines the stored procedure. Each stored procedure-type dataset does this differently, but typically you specify a database connection component:
   - For *TStoredProc*, specify a *TDatabase* component or a BDE alias using the *DatabaseName* property.
   - For *TADOStoredProc*, specify a *TADOConnection* component using the *Connection* property.
   - For *TSQLStoredProc*, specify a *TSQLConnection* component using the *SQLConnection* property.
   - For *TIBStoredProc*, specify a *TIBConnection* component using the *Database* property.

   For information about using database connection components, see Chapter 17, "Connecting to databases".
3. Specify the stored procedure to execute. For most stored procedure-type datasets, you do this by setting the *StoredProcName* property. The one exception is *TADOStoredProc*, which has a *ProcedureName* property instead.
4. If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the stored procedure-type dataset. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Provide input parameter values for the stored procedure, if necessary. If the server does not provide information about all stored procedure parameters, you may need to provide additional input parameter information, such as parameter names and data types. For information about working with stored procedure parameters, see "Working with stored procedure parameters" on page 18-50.
6. Execute the stored procedure. For stored procedures that return a cursor, use the *Active* property or the *Open* method. To execute stored procedures that do not return any results or that only return output parameters, use the *ExecProc* method at runtime. If you plan to execute the stored procedure more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the stored procedure. For information about preparing a query, see "Executing stored procedures that don't return a result set" on page 18-53.
7. Process any results. These results can be returned as result and output parameters, or they can be returned as a result set that populates the stored procedure-type dataset. Some stored procedures return multiple cursors. For details on how to access the additional cursors, see "Fetching multiple result sets" on page 18-53.

Working with stored procedure parameters

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.
- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- A *result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, on MS-SQL Server and Sybase stored procedures always return a result parameter, but the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by the *Params* property (in *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*) or the *Parameters* property (in *TADOStoredProc*). When you assign a value to the *StoredProcName* (or *ProcedureName*) property, the dataset automatically generates objects for each parameter of the stored procedure. For some datasets, if the stored procedure name is not specified until runtime, objects for each parameter must be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* or *TParameter* objects allows a single dataset to be used with any number of available stored procedures.

Note Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters.

Setting up parameters at design time

You can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the Object Inspector.

Important You can assign values to input parameters by selecting them in the parameter collection editor and using the Object Inspector to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add, you must fully describe the parameter. Even if you do not need to add any parameters, you should check the properties of individual parameter objects to ensure that they are correct.

If the dataset has a *Params* property (*TParam* objects), the following properties must be correctly specified:

- The *Name* property indicates the name of the parameter as it is defined by the stored procedure.
- The *DataType* property gives the data type for the parameter's value. When using *TSQLStoredProc*, some data types require additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *ParamType* property indicates the type of the selected parameter. This can be *ptInput* (for input parameters), *ptOutput* (for output parameters), *ptInputOutput* (for input/output parameters) or *ptResult* (for result parameters).
- The *Value* property specifies a value for the selected parameter. You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

If the dataset uses a *Parameters* property (*TParameter* objects), the following properties must be correctly specified:

- The *Name* property indicates the name of the parameter as it is defined by the stored procedure.
- The *DataType* property gives the data type for the parameter's value. For some data types, you must provide additional information:
  - The *NumericScale* property indicates the number of decimal places for numeric parameters.
  - The *Precision* property indicates the total number of digits for numeric parameters.
  - The *Size* property indicates the number of characters in string parameters.
- The *Direction* property gives the type of the selected parameter. This can be *pdInput* (for input parameters), *pdOutput* (for output parameters), *pdInputOutput* (for input/output parameters) or *pdReturnValue* (for result parameters).
- The *Attributes* property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The *Value* property specifies a value for the selected parameter. Do not set values for output and result parameters. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

Using parameters at runtime

With some datasets, if the name of the stored procedure is not specified until runtime, no *TParam* objects are automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method:

```
var
  P1, P2: TParam;
begin
  ...
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := ‘EMP_NO’;
      Params[1].Name := ‘PROJ_ID’;
      ParamByname(‘EMP_NO’).AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname(‘PROJ_ID’).AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ...
end;
```

Even if you do not need to add the individual parameter objects at runtime, you may want to access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. You can use the dataset’s *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/output parameter, executes the stored procedure, and retrieves the returned value:

```
with SQLStoredProc1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Preparing stored procedures

As with query-type datasets, stored procedure-type datasets must be prepared before they execute the stored procedure. Preparing a stored procedure tells the data access layer and the database server to allocate resources for the stored procedure and to bind parameters. These operations can improve performance.

If you attempt to execute a stored procedure before preparing it, the dataset automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to explicitly prepare it by setting the Prepared property to *True*.

```
MyProc.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the stored procedure are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change the parameters when using Oracle overloaded procedures).

Executing stored procedures that don't return a result set

When a stored procedure returns a cursor, you execute it the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often stored procedures do not return any data, or only return results in output parameters. You can execute a stored procedure that does not return a result set by calling *ExecProc*. After executing the stored procedure, you can use the *ParamByName* method to read the value of the result parameter or of any output parameters:

```
MyStoredProcedure.ExecProc;  { does not return a result set }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
```

Note *TADOStoredProc* does not have a *ParamByName* method. To obtain output parameter values when using ADO, access parameter objects using the *Parameters* property.

Tip If you are executing the procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

Fetching multiple result sets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. If you are using *TSQLStoredProc* or *TADOStoredProc*, you can access the other sets of records by calling the *NextRecordSet* method:

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

In *TSQLStoredProc, NextRecordSet* returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. In *TADOStoredProc*, *NextRecordset* returns an interface that can be assigned to the *RecordSet* property of an existing ADO dataset. For either class, the method returns the number of records in the returned dataset as an output parameter.

The first time you call *NextRecordSet*, it returns the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional cursors, *NextRecordSet* returns **nil**.

Chapter

19

Working with field components

This chapter describes the properties, events, and methods common to the *TField* object and its descendants. Field components represent individual fields (columns) in datasets. This chapter also describes how to use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a *TField* object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as *TDBEdit* and *TDBGrid* access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

Table 19.1 TFloatField properties that affect data display

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications. The following sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

Dynamic field components

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact *TField* descendant created for each column is determined by field type information received from the database or (for *TClientDataSet*) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool such as SQL explorer, you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application:

1. Place datasets and data sources in a data module.
2. Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
3. Associate the data sources with the datasets.
4. Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any field name you specify will exist when the dataset is opened.
5. Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

Persistent field components

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.
- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

Note When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see “Dynamic field components” on page 19-2.

Note One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see “Creating a customized grid” on page 15-16.

Creating persistent fields

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset:

1. Place a dataset in a data module.
2. Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, If you are using *TADODataSet*, you can set the *Connection* property to a properly configured *TADOConnection* component and set the *CommandText* property to a valid query.
3. Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

   The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

   Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed.

   The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.
4. Choose Add Fields from the Fields editor context menu.
5. Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open the dataset.

Arranging persistent fields

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields:

1. Select the fields. You can select and order one or more fields at a time.
2. Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use *Ctrl+Up* and *Ctrl+Dn* to change an individual field's order in the list.

Defining new persistent fields

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's *Name* property and is only provided for informational purposes (*Name* is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.

- The Type combo box in the Field properties group lets you specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

Table 19.2 Special persistent field kinds

Field kind	Purpose
Data	Replaces an existing field (for example to change its data type)
Calculated	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify. (not supported by unidirectional datasets)
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Aggregate	Displays a value summarizing the data in a set of records from a client dataset.

The Lookup definition group box is only used to create lookup fields. This is described more fully in "Defining a lookup field" on page 19-8.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset, follow these steps:

1. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
2. In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.

4 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

5 Select Data in the Field type radio group if it is not already selected.

6 Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 19-10.

Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1 Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.

2 Choose a data type for the field from the Type combo box.

3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.

5 Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's **type** declaration.

6 Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field" on page 19-7.

Note To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 19-10.

Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

1 Select the dataset component from the Object Inspector drop-down list.

2 Choose the Object Inspector Events page.

3 Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the dataset component.

4 Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
  + ' ' + CustomersZip.Value;
```

Note When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's *State* property and only recomputing the value when *State* is *dsInternalCalc*. See "Using internally calculated fields in client datasets" on page 23-10 for details.

Defining a lookup field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

Note Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box:

1 Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.

2 Choose a data type for the field from the Type combo box.

3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.

5 Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.

6 Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.

7 Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.

8 Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the *LookupCache* property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to *True* to cache the values of a lookup field when the *LookupDataSet* is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields* values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

Tip You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Be sure that the *LookupDataSet* property is nil. Then, use the *LookupList* property's *Add* method to fill it with lookup values. Set the *LookupCache* property to *True*. The field will use the supplied lookup list without overwriting it with values from a lookup dataset.

If every record of *DataSet* has different values for *KeyFields*, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call *RefreshLookupList* to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the *LookupResultField* for every set of *LookupKeyFields* values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

Defining an aggregate field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See "Using maintained aggregates" on page 23-11 for details about maintained aggregates.

To create an aggregate field in the New Field dialog box:

1. Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
2. Choose aggregate data type for the field from the Type combo box.
3. Select Aggregate in the Field type radio group.
4. Choose OK. The newly defined aggregate field is automatically added to the client dataset and its *Aggregates* property is automatically updated to include the appropriate aggregate specification.
5. Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see "Specifying aggregates" on page 23-11.

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the underlying client data set.

Deleting persistent field components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table. To remove one or more persistent field components for a dataset:

1. Select the field(s) to remove in the Fields editor list box.
2. Press *Del*.

Note You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see "Creating persistent fields" on page 19-4.

Note If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

Setting persistent field properties and events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

Setting display and edit properties at design time

To edit the display properties of a selected field component, switch to the Properties page on the Object Inspector window. The following table summarizes display properties that can be edited.

Table 19.3 Field component properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field contents within a data-aware component.
ConstraintErrorMessage	Specifies the text to display when edits clash with a constraint condition.
CustomConstraint	Specifies a local constraint to apply to data during editing.
Currency	Numeric fields only. *True*: displays monetary values. *False* (default): does not display monetary values.
DisplayFormat	Specifies the format of data displayed in a data-aware component.
DisplayLabel	Specifies the column name for a field in a data-aware grid component.
DisplayWidth	Specifies the width, in characters, of a grid column that display this field.
EditFormat	Specifies the edit format of data in a data-aware component.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
FieldKind	Specifies the type of field to create.
FieldName	Specifies the actual name of a column in the table from which the field derives its value and data type.
HasConstraints	Indicates whether there are constraint conditions imposed on a field.
ImportedConstraint	Specifies an SQL constraint imported from the Data Dictionary or an SQL server.
Index	Specifies the order of the field in a dataset.
LookupDataSet	Specifies the table used to look up field values when *Lookup* is *True*.
LookupKeyFields	Specifies the field(s) in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset from which to copy values into this field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the component name of the field component within Delphi.
Origin	Specifies the name of the field as it appears in the underlying database.
Precision	Numeric fields only. Specifies the number of significant digits.

Table 19.3 Field component properties (continued)

Property	Purpose
ReadOnly	*True*: Displays field values in data-aware controls, but prevents editing. *False* (the default): Permits display and editing of field values.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields.
Tag	Long integer bucket available for programmer use in every component as needed.
Transliterate	*True* (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. *False*: Locale translation does not occur.
Visible	*True* (the default): Permits display of field in a data-aware grid. *False*: Prevents display of field in a data-aware grid component. User-defined components can make display decisions based on this property.

Not all properties are available for all field components. For example, a field component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type *TFloatField* does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see "Controlling and masking user input" on page 19-14.

Setting field component properties at runtime

You can use and manipulate the properties of field component at runtime. Access persistent field components by name, where the name can be obtained by concatenating the field name to the dataset name.

For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly := True;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
```

Creating attribute sets for field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

Note Attribute sets and the Data Dictionary are only available for BDE-enabled datasets.

To create an attribute set based on a field component in a dataset:

1 Double-click the dataset to invoke the Fields editor.

2 Select the field for which to set properties.

3 Set the desired properties for the field in the Object Inspector.

4 Right-click the Fields editor list box to invoke the context menu.

5 Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save Attributes As instead of Save Attributes from the context menu.

Once you have created a new attribute set and added it to the Data Dictionary, you can then associate it with other persistent field components. Even if you later remove the association, the attribute set remains defined in the Data Dictionary.

Note You can also create attribute sets directly from the SQL Explorer. When you create an attribute set using SQL Explorer, it is added to the Data Dictionary, but not applied to any fields. SQL Explorer lets you specify two additional attributes: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) that is automatically placed on a form when a field based on the attribute set is dragged to the form. For more information, see the online help for the SQL Explorer.

Associating attribute sets with field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1 Double-click the dataset to invoke the Fields editor.

2 Select the field for which to apply an attribute set.

3 Invoke the context menu and choose Associate Attributes.

4 Select or enter the attribute set to apply from the Associate Attributes dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

Important If the attribute set in the Data Dictionary is changed at a later date, you must reapply the attribute set to each field component that uses it. You can invoke the Fields editor and multi-select field components within a dataset when reapplying attributes.

Removing attribute associations

If you change your mind about associating an attribute set with a field, you can remove the association by following these steps:

1 Invoke the Fields editor for the dataset containing the field.

2 Select the field or fields from which to remove the attribute association.

3 Invoke the context menu for the Fields editor and choose Unassociate Attributes.

Important Unassociating an attribute set does not change any field properties. A field retains the settings it had when the attribute set was applied to it. To change these properties, select the field in the Fields editor and set its properties in the Object Inspector.

Controlling and masking user input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, and *TDateTimeField*, and *TSQLTimeStampField* components. You can use existing masks or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the Object Inspector.

Note For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component:

1 Select the component in the Fields editor or Object Inspector.

2 Click the Properties page in the Object Inspector.

3 Double-click the values column for the EditMask field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

Using default formatting for numeric, date, and time fields

Delphi provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, and *TTimeField*, and *TSQLTimeStampField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Table 19.4 Field component formatting routines

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*,
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField*, *TBCDField*
BcdToStrF	*TFMTBcdField*

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Control Panel. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the *EditFormat* is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and *EditFormat* properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime.

Handling events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

Table 19.5 Field component events

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component:

1. Select the component.
2. Select the Events page in the Object Inspector.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

Working with field component methods at runtime

Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see the entries for *TField* and its descendants in the online *VCL Reference*.

Table 19.6 Selected field component methods

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

Displaying, converting, and accessing field values

Data-aware controls such as *TDBEdit* and *TDBGrid* automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Chapter 15, "Using data controls."

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls, your application is responsible for tracking when to update controls because field values change. If the dataset has a datasource component, you can use its events to help you do this. In particular, the *OnDataChange* event lets you know when you may need to update a control's value and the *OnStateChange* event can help you determine when to enable or disable controls. For more information on these events, see "Responding to changes mediated by the data source" on page 15-4.

The following topics discuss how to work with field values so that you can display them in standard controls.

Displaying field component values in standard controls

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a *TEdit* control because the value of the *CustomersCompany* field may have changed:

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
  Edit3.Text := CustomersCompany.Value;
end;
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in properties for handling conversions.

Note You can also use Variants to access and set field values. For more information about using variants to access and set field values, see "Accessing field values with the default dataset property" on page 19-19.

Converting field values

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations.

The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		
TWordField	yes	yes	yes	yes		
TLargeintField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		
TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes		yes	yes	
TDateField	yes	yes		yes	yes	
TTimeField	yes	yes		yes	yes	
TSQLTimeStampField	yes	yes		yes	yes	
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any data types not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. Table 19.7 lists permissible conversions that produce special results:

Table 19.7 Special conversion results

Conversion	Result
String to Boolean	Converts “True,” “False,” “Yes,” and “No” to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime or SQLTimeStamp to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to “True” or “False.”

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

Accessing field values with the default dataset property

The most general method for accessing a field’s value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Because the *FieldValues* property is of type Variant, it automatically converts other datatypes into a Variant value.

For more information about Variants, see the online help.

Accessing field values with a dataset’s Fields property

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each field component's conversion properties. For more information about field component conversion properties, see "Converting field values" on page 19-17.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

Accessing field values with a dataset's FieldByName method

You can also access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

Setting a default value for a field

You can specify how a default value for a field in a client dataset or a BDE-enabled dataset should be calculated at runtime using the *DefaultExpression* property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

Note If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the dataset posts the record containing the field, before the edited record is applied to the database server.

Working with constraints

Field components in client datasets or BDE-enabled datasets can use SQL server constraints. In addition, your applications can create and use custom constraints for these datasets that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store.

Creating a custom constraint

A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a prevalidation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

CustomConstraint is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

Note Custom constraints are only available in BDE-enabled and client datasets.

Custom constraints are imposed in addition to any constraints to the field's value that come from the server. To see the constraints imposed by the server, read the *ImportedConstraint* property.

Using server constraints

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. While you could replicate such conditions in your client applications, client datasets and BDE-enabled datasets offer the *ImportedConstraint* property to propagate a server's constraints locally.

ImportedConstraint is a read-only property that specifies an SQL clause that limits field values in some manner. For example:

```
Value > 0 and Value < 100
```

Do not change the value of *ImportedConstraint*, except to edit nonstandard or server-specific SQL that has been imported as a comment because it cannot be interpreted by the database engine.

To add additional constraints on the field value, use the *CustomConstraint* property. Custom constraints are imposed in addition to the imported constraints. If the server constraints change, the value of *ImportedConstraint* also changed but constraints introduced in the *CustomConstraint* property persist.

Removing constraints from the *ImportedConstraint* property will not change the validity of field values that violate those constraints. Removing constraints results in the constraints being checked by the server instead of locally. When constraints are checked locally, the error message supplied as the *ConstraintErrorMessage* property is displayed when violations are found, instead of displaying an error message from the server.

Using object fields

Object fields are fields that represent a composite of other, simpler data types. These include ADT (Abstract Data Type) fields, Array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields are fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

Table 19.8 Types of object field components

Component name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's *ObjectView* property to *True*, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

Table 19.9 Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classifies the object field.
FieldCount	Number of child fields belonging to the object field.
FieldValues	Provides access to the values of the child fields.

Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as *TDBEdit* that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed an edited just like any other normal data field.

A *TDBGrid* control displays ADT and array field data differently, depending on the value of the dataset's *ObjectView* property. When *ObjectView* is *False*, each child field appears in a single column. When *ObjectView* is *True*, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma-delimited string containing the child fields.

Working with ADT fields

ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called *CityEdit*, and use the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the *Customer* table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
CityEdit.Text := CustomerAddrCity.AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's *ObjectView* property to *True*.

Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's *FieldByName* method by qualifying the name of the child field with the ADT field's name:

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

Using the deteset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
CityEdit.Text := Customer['Address.City'];
```

Note that you can omit the property name (*FieldValues*) because *FieldValues* is the dataset's default property.

Note Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is *False*.

Using the ADT field's FieldValues property

You can access the value of a child field with the *TADTField*'s *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

Because *FieldValues* is the default property of *TADTField*, the property name (*FieldValues*) can be omitted. Thus, the following statement is equivalent to the one above:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

Using the ADT field's Fields property

Each ADT field has a *Fields* property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

or by name:

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

Working with array fields

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The SparseArrays property of *TDataSet* determines whether a unique *TField* object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's *ObjectView* property must be set to *True* before you can access the elements of an array field.

Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component represent the *TelNos_Array* field and its six elements:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

Because *FieldValues* is the default property of *TArrayField*, this can also be written

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

Using the array field's Fields property

TArrayField has a *Fields* property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

Working with dataset fields

Dataset fields provide access to data stored in a nested dataset. The NestedDataSet property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.

Displaying dataset fields

TDBGrid controls enable the display of data stored in data set fields. In a *TDBGrid* control, a dataset field is indicated in each cell of a dataset column with the string "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a dataset field, the following code will display the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested data set is just that, a data set, the means to get at its data is via a *TDataSet* descendant. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a BDE-enabled dataset uses *TNestedTable* to represent the data in its dataset fields, while client datasets use other client datasets.

To access the data in a dataset field,

1 Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.

2 Create a dataset to represent the values in that dataset field. It must be of a type compatible with the parent dataset.

3 Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Before inserting records into a nested dataset, you should be sure to post the corresponding record in the master table, if it has just been inserted. If the inserted record is not posted, it will be automatically posted before the nested dataset posts.

Working with reference fields

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the *Fields* property on the *TReferenceField*.

Displaying reference fields

In a *TDBGrid* control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset:

1. Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the value of that dataset field.
3. Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's Fields property to access the data in a reference field. For example, the following lines are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

When data in a reference field is edited, it is actually the referenced data that is modified.

To assign a reference field, you need to first use a SELECT statement to select the reference from the table, and then assign. For example:

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := ‘SELECT REF(A) FROM AddressTable A WHERE A.City = ‘’San
  Francisco’’’;
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

Chapter

20

Using the Borland Database Engine

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. Depending on your version of Delphi, you can use the drivers for local databases (Paradox, dBASE, FoxPro, and Access), SQL Links drivers for remote database servers such as InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2, and an ODBC adapter that lets you supply your own ODBC drivers.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broad range of support for database manipulation. Although you can use the BDE's API directly in your application, the components on the BDE page of the Component palette wrap most of this functionality for you.

Note For information on the BDE API, see its online help file, BDE32.hlp, which is installed in the directory where you install the Borland Database Engine.

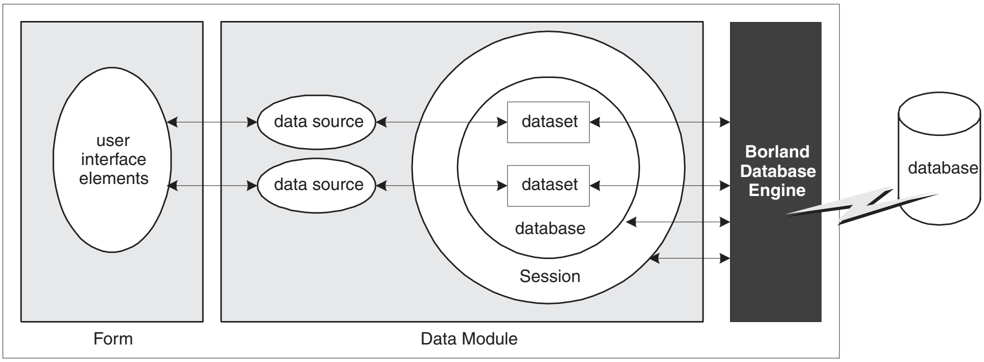
BDE-based architecture

When using the BDE, your application uses a variation of the general database architecture described in "Database architecture" on page 14-5. In addition to the user interface elements, datasource, and datasets common to all Delphi database applications, A BDE-based application can include

- One or more database components to control transactions and to manage database connections.
- One or more session components to isolate data access operations such as database connections, and to manage groups of databases.

The relationships between the components in a BDE-based application are illustrated in Figure 20.1:

Figure 20.1 Components in a BDE-based application



Using BDE-enabled datasets

BDE-enabled datasets use the Borland Database Engine (BDE) to access data. They inherit the common dataset capabilities described in Chapter 18, "Understanding datasets," using the BDE to provide the implementation. In addition, all BDE datasets add properties, events, and methods for

- Associating a dataset with database and session connections.
- Caching BLOBs.
- Obtaining a BDE handle.

There are three BDE-enabled datasets:

- *TTable*, a table-type dataset that represents all of the rows and columns of a single database table. See "Using table-type datasets" on page 18-24 for a description of features common to table-type datasets. See "Using TTable" on page 20-4 for a description of features unique to *TTable*.
- *TQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See "Using query-type datasets" on page 18-41 for a description of features common to query-type datasets. See "Using TQuery" on page 20-8 for a description of features unique to *TQuery*.
- *TStoredProc*, a stored procedure-type dataset that executes a stored procedure that is defined on a database server. See "Using stored procedure-type datasets" on page 18-48 for a description of features common to stored procedure-type datasets. See "Using TStoredProc" on page 20-11 for a description of features unique to *TStoredProc*.

Note In addition to the three types of BDE-enabled datasets, there is a BDE-based client dataset (*TBDEClientDataSet*) that can be used for caching updates. For information on caching updates, see "Using a client dataset to cache updates" on page 23-15.

Associating a dataset with database and session connections

In order for a BDE-enabled dataset to fetch data from a database server it needs to use both a database and a session.

- Databases represent connections to specific database servers. The database identifies a BDE driver, a particular database server that uses that driver, and a set of connection parameters for connecting to that database server. Each database is represented by a *TDatabase* component. You can either associate your datasets with a *TDatabase* component you add to a form or data module, or you can simply identify the database server by name and let Delphi generate an implicit database component for you. Using an explicitly-created *TDatabase* component is recommended for most applications, because the database component gives you greater control over how the connection is established, including the login process, and lets you create and use transactions.

  To associate a BDE-enabled dataset with a database, use the *DatabaseName* property. DatabaseName is a string that contains different information, depending on whether you are using an explicit database component and, if not, the type of database you are using:

  - If you are using an explicit *TDatabase* component, *DatabaseName* is the value of the *DatabaseName* property of the database component.
  - If you are want to use an implicit database component and the database has a BDE alias, you can specify a BDE alias as the value of *DatabaseName*. A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on). Use the BDE Administration tool or the SQL explorer to create and manage BDE aliases.
  - If you want to use an implicit database component for a Paradox or dBASE database, you can also use *DatabaseName* to simply specify the directory where the database tables are located.

- A session provides global management for a group of database connections in an application. When you add BDE-enabled datasets to your application, your application automatically contains a session component, named *Session*. As you add database and dataset components to the application, they are automatically associated with this default session. It also controls access to password protected Paradox files, and it specifies directory locations for sharing Paradox files over a network. You can control database connections and access to Paradox files using the properties, events, and methods of the session.

  You can use the default session to control all database connections in your application. Alternatively, you can add additional session components at design time or create them dynamically at runtime to control a subset of database connections in an application. To associate your dataset with an explicitly created session component, use the *SessionName* property. If you do not use explicit session components in your application, you do not have to provide a value for this property. Whether you use the default session or explicitly specify a session using the *SessionName* property, you can access the session associated with a dataset by reading the *DBSession* property.

Note If you use a session component, the *SessionName* property of a dataset must match the *SessionName* property for the database component with which the dataset is associated.

For more information about *TDatabase* and *TSession*, see “Connecting to databases with TDatabase” on page 20-12 and “Managing database sessions” on page 20-16.

Caching BLOBs

BDE-enabled datasets all have a *CacheBlobs* property that controls whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records.

In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to *False* to ensure that your application always sees the latest version of a BLOB field.

Obtaining a BDE handle

You can use BDE-enabled datasets without ever needing to make direct API calls to the Borland Database Engine. The BDE-enabled datasets, in combination with database and session components, encapsulate much of the BDE functionality. However, if you need to make direct API calls to the BDE, you may need BDE handles for resources managed by the BDE. Many BDE APIs require these handles as parameters.

All BDE-enabled datasets include three read-only properties for accessing BDE handles at runtime:

- *Handle* is a handle to the BDE cursor that accesses the records in the dataset.
- *DBHandle* is a handle to the database that contains the underlying tables or stored procedure.
- *DBLocale* is a handle to the BDE language driver for the dataset. The locale controls the sort order and character set used for string data.

These properties are automatically assigned to a dataset when it is connected to a database server through the BDE. For more information about the BDE API, see the online help file, BDE32.HLP.

Using TTable

TTable encapsulates the full structure of and data in an underlying database table. It implements all of the basic functionality introduced by *TDataSet*, as well as all of the special features typical of table-type datasets. Before looking at the unique features introduced by *TTable*, you should familiarize yourself with the common database features described in “Understanding datasets,” including the section on table-type datasets that starts on page 18-24.

Because *TTable* is a BDE-enabled dataset, it must be associated with a database and a session. "Associating a dataset with database and session connections" on page 20-3 describes how you form these associations. Once the dataset is associated with a database and session, you can bind it to a particular database table by setting the *TableName* property and, if you are using a Paradox, dBASE, FoxPro, or comma-delimited ASCII text table, the *TableType* property.

Note The table must be closed when you change its association to a database, session, or database table, or when you set the *TableType* property. However, before you close the table to change these properties, first post or discard any pending changes. If cached updates are enabled, call the *ApplyUpdates* method to write the posted changes to the database.

TTable components are unique in the support they offer for local database tables (Paradox, dBASE, FoxPro, and comma-delimited ASCII text tables). The following topics describe the special properties and methods that implement this support.

In addition, *TTable* components can take advantage of the BDE's support for batch operations (table level operations to append, update, delete, or copy entire groups of records). This support is described in "Importing data from another table" on page 20-8.

Specifying the table type for local tables

If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the *TableType* property to determine the table's type (its expected structure). *TableType* is not used when *TTable* represents an SQL-based table on a database server.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its filename extension. Table 20.1 summarizes the file extensions recognized by the BDE and the assumptions it makes about a table's type:

Table 20.1 Table types recognized by the BDE based on file extension

Extension	Table type
No file extension	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII text

If your local Paradox, dBASE, and ASCII text tables use the file extensions as described in Table 20.1, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. Table 20.2 indicates the values you can assign to *TableType*:

Table 20.2 TableType values

Value	Table type
ttDefault	Table type determined automatically by the BDE
ttParadox	Paradox

Table 20.2 TableType values (continued)

Value	Table type
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Comma-delimited ASCII text

Controlling read/write access to local tables

Like any table-type dataset, *TTable* lets you control read and write access by your application using the *ReadOnly* property.

In addition, for Paradox, dBASE, and FoxPro tables, *TTable* can let you control read and write access to tables by other applications. The *Exclusive* property controls whether your application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. Your request for exclusive access is not honored if the table is already in use when you attempt to open it.

The following statements open a table for exclusive access:

```
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
```

Note You can attempt to set *Exclusive* on SQL tables, but some servers do not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

Specifying a dBASE index file

For most servers, you use the methods common to all table-type datasets to specify an index. These methods are described in "Sorting records with indexes" on page 18-25.

For dBASE tables that use non-production index files or dBASE III PLUS-style indexes (*.NDX), however, you must use the *IndexFiles* and *IndexName* properties instead. Set the *IndexFiles* property to the name of the non-production index file or list the .NDX files. Then, specify one index in the *IndexName* property to have it actively sorting the dataset.

At design time, click the ellipsis button in the *IndexFiles* property value in the Object Inspector to invoke the Index Files editor. To add one non-production index file or .NDX file: click the Add button in the Index Files dialog and select the file from the Open dialog. Repeat this process once for each non-production index file or .NDX file. Click the OK button in the Index Files dialog after adding all desired indexes.

This same operation can be performed programmatically at runtime. To do this, access the *IndexFiles* property using properties and methods of string lists. When adding a new set of indexes, first call the *Clear* method of the table's *IndexFiles* property to remove any existing entries. Call the *Add* method to add each non-production index file or .NDX file:

```
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
```

After adding any desired non-production or .NDX index files, the names of individual indexes in the index file are available, and can be assigned to the *IndexName* property. The index tags are also listed when using the *GetIndexNames* method and when inspecting index definitions through the *TIndexDef* objects in the *IndexDefs* property. Properly listed .NDX files are automatically updated as data is added, changed, or deleted in the table (regardless of whether a given index is used in the *IndexName* property).

In the example below, the *IndexFiles* for the *AnimalsTable* table component is set to the non-production index file ANIMALS.MDX, and then its *IndexName* property is set to the index tag called “NAME”:

```
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
```

Once you have specified the index file, using non-production or .NDX indexes works the same as any other index. Specifying an index name sorts the data in the table and makes it available for indexed-based searches, ranges, and (for non-production indexes) master-detail linking. See “Using table-type datasets” on page 18-24 for details on these uses of indexes.

There are two special considerations when using dBASE III PLUS-style .NDX indexes with *TTable* components. The first is that .NDX files cannot be used as the basis for master-detail links. The second is that when activating a .NDX index with the *IndexName* property, you must include the .NDX extension in the property value as part of the index name:

```
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
```

Renaming local tables

To rename a Paradox or dBASE table at design time, right-click the table component and select Rename Table from the context menu.

To rename a Paradox or dBASE table at runtime, call the table’s *RenameTable* method. For example, the following statement renames the Customer table to CustInfo:

```
Customer.RenameTable(‘CustInfo’);
```

Importing data from another table

You can use a table component's *BatchMove* method to import data from another table. *BatchMove* can

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

BatchMove takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. Table 20.3 describes the possible settings for the mode specification:

Table 20.3 BatchMove import modes

Value	Meaning
batAppend	Append all records from the source table to the end of this table.
batAppendUpdate	Append all records from the source table to the end of this table and update existing records in this table with matching records from the source table.
batCopy	Copy all records from the source table into this table.
batDelete	Delete all records in this table that also appear in the source table.
batUpdate	Update existing records in this table with matching records from the source table.

For example, the following code updates all records in the current table with records from the *Customer* table that have the same values for fields in the current index:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

BatchMove returns the number of records it imports successfully.

Caution Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

BatchMove performs only some of the batch operations supported by the BDE. Additional functions are available using the *TBatchMove* component. If you need to move a large amount of data between or among tables, use *TBatchMove* instead of calling a table's *BatchMove* method. For information about using *TBatchMove*, see "Using TBatchMove" on page 20-47.

Using TQuery

TQuery represents a single Data Definition Language (DDL) or Data Manipulation Language (DML) statement (For example, a SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, or ALTER TABLE command). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. *TQuery* implements all of the basic functionality introduced by *TDataSet*, as well as all of the special features typical of query-type datasets. Before looking at the unique features introduced by *TQuery*, you should familiarize yourself with the common database features described in "Understanding datasets," including the section on query-type datasets that starts on page 18-41.

Because *TQuery* is a BDE-enabled dataset, it must usually be associated with a database and a session. (The one exception is when you use the *TQuery* for a heterogeneous query.) "Associating a dataset with database and session connections" on page 20-3 describes how you form these associations. You specify the SQL statement for the query by setting the *SQL* property.

A TQuery component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL is a subset of the SQL-92 specification. Most DML is supported and enough DDL syntax to work with these types of tables. See the local SQL help, LOCALSQL.HLP, for details on supported SQL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase's SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference.*
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase. You must install the appropriate SQL Link driver and client software (vendor-supplied) specific to the database server to access a remote server. Any standard SQL syntax supported by these servers is allowed. For information on SQL syntax, limitations, and extensions, see the documentation for your particular server.

Creating heterogeneous queries

TQuery supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query, follow these steps:

1. Define separate BDE aliases for each database accessed in the query using the BDE BDE Administration tool or the SQL explorer.
2. Leave the *DatabaseName* property of the *TQuery* blank; the names of the databases used will be specified in the SQL statement.
3. In the SQL property, specify the SQL statement to execute. Precede each table name in the statement with the BDE alias for the table's database, enclosed in colons. This whole reference is then enclosed in quotation marks.
4. Set any parameters for the query in the *Params* property.
5. Call *Prepare* to prepare the query for execution prior to executing it for the first time.
6. Call *Open* or *ExecSQL* depending on the type of query you are executing.

For example, suppose you define an alias called *Oracle1* for an Oracle database that has a CUSTOMER table, and *Sybase1* for a Sybase database that has an ORDERS table. A simple query against these two tables would be:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

As an alternative to using a BDE alias to specify the database in a heterogeneous query, you can use a *TDatabase* component. Configure the *TDatabase* as normal to point to the database, set the *TDatabase.DatabaseName* to an arbitrary but unique value, and then use that value in the SQL statement instead of a BDE alias name.

Obtaining an editable result set

To request a result set that users can edit in data-aware controls, set a query component's *RequestLive* property to *True*. Setting *RequestLive* to *True* does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether the query uses the local SQL parser or a server's SQL parser.

- Queries where table names are preceded by a BDE database alias (as in heterogeneous queries) and queries executed against Paradox or dBASE are parsed by the BDE using Local SQL. When queries use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. When using Local SQL, a live result set for a query against a single table or view is returned if the query does not contain any of the following:
  - DISTINCT in the SELECT clause
  - Joins (inner, outer, or UNION)
  - Aggregate functions with or without GROUP BY or HAVING clauses
  - Base tables or views that are not updatable
  - Subqueries
  - ORDER BY clauses not based on an index
- Queries against a remote database server are parsed by the server. If the *RequestLive* property is set to *True*, the SQL statement must abide by Local SQL standards in addition to any server-imposed restrictions because the BDE needs to use it for conveying data changes to the table. A live result set for a query against a single table or view is returned if the query does not contain any of the following:
  - A DISTINCT clause in the SELECT statement
  - Aggregate functions, with or without GROUP BY or HAVING clauses
  - References to more than one base table or updatable views (joins)
  - Subqueries that reference the table in the FROM clause or other tables

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *True*. Even if the query returns a live result set, you may not be able to update the result set directly if it contains linked fields or you switch indexes before attempting an update. If these conditions exist, you should treat the result set as a read-only result set, and update it accordingly.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for SQL queries made against a remote server.

Updating read-only result sets

Applications can update data returned in a read-only result set if they are using cached updates.

If you are using a client dataset to cache updates, the client dataset or its associated provider can automatically generate the SQL for applying updates unless the query represents multiple tables. If the query represents multiple tables, you must indicate how to apply the updates:

- If all updates are applied to a single database table, you can indicate the underlying table to update in an *OnGetTableName* event handler.
- If you need more control over applying updates, you can associate the query with an update object (*TUpdateSQL*). A provider automatically uses this update object to apply updates:
  1. Associate the update object with the query by setting the query's *UpdateObject* property to the *TUpdateSQL* object you are using.
  2. Set the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties to SQL statements that perform the appropriate updates for your query's data.

If you are using the BDE to cache updates, you must use an update object.

Note For more information on using update objects, see "Using update objects to update a dataset" on page 20-39.

Using TStoredProc

TStoredProc represents a stored procedure. It implements all of the basic functionality introduced by *TDataSet*, as well as most of the special features typical of stored procedure-type datasets. Before looking at the unique features introduced by *TStoredProc*, you should familiarize yourself with the common database features described in "Understanding datasets," including the section on stored procedure-type datasets that starts on page 18-48.

Because *TStoredProc* is a BDE-enabled dataset, it must be associated with a database and a session. "Associating a dataset with database and session connections" on page 20-3 describes how you form these associations. Once the dataset is associated with a database and session, you can bind it to a particular stored procedure by setting the *StoredProcName* property.

TStoredProc differs from other stored procedure-type datasets in the following ways:

- It gives you greater control over how to bind parameters.
- It provides support for Oracle overloaded stored procedures.

Binding parameters

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

TStoredProc lets you use the *ParamBindMode* property to specify how parameters should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters.

Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set *ParamBindMode* to *pbByNumber*.

Tip If you want to set *ParamBindMode* to *pbByNumber*, you need to specify the correct parameter types in the correct order. You can view a server's stored procedure source code in the SQL Explorer to determine the correct order and type of parameters to specify.

Working with Oracle overloaded stored procedures

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's *Overload* property enables an application to specify the procedure to execute.

If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on.

Note Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.

Connecting to databases with TDatabase

When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a *TDatabase* component. A database component represents the connection to a single database in the context of a BDE session.

TDatabase performs many of the same tasks as and shares many common properties, methods, and events with other database connection components. These commonalities are described in Chapter 17, "Connecting to databases."

In addition to the common properties, methods, and events, *TDatabase* introduces many BDE-specific features. These features are described in the following topics.

Associating a database component with a session

All database components must be associated with a BDE session. Use the *SessionName*, establish this association. When you first create a database component at design time, *SessionName* is set to "Default", meaning that it is associated with the default session component that is referenced by the global *Session* variable.

Multi-threaded or reentrant BDE applications may require more than one session. If you need to use multiple sessions, add *TSession* components for each session. Then, associate your dataset with a session component by setting the *SessionName* property to a session component's *SessionName* property.

At runtime, you can access the session component with which the database is associated by reading the *Session* property. If *SessionName* is blank or "Default", then the *Session* property references the same *TSession* instance referenced by the global *Session* variable. *Session* enables applications to access the properties, methods, and events of a database component's parent session component without knowing the session's actual name.

For more information about BDE sessions, see "Managing database sessions" on page 20-16.

If you are using an implicit database component, the session for that database component is the one specified by the dataset's *SessionName* property.

Understanding database and session component interactions

In general, session component properties provide global, default behaviors that apply to all implicit database components created at runtime. For example, the controlling session's *KeepConnections* property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPassword* event for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box.

Session methods apply somewhat differently. *TSession* methods affect all database components, regardless of whether they are explicitly created or instantiated implicitly by a dataset. For example, the session method *DropConnections* closes all datasets belonging to a session's database components, and then drops all database connections, even if the *KeepConnection* property for individual database components is *True*.

Database component methods apply only to the datasets associated with a given database component. For example, suppose the database component *Database1* is associated with the default session. *Database1.CloseDataSets()* closes only those datasets associated with *Database1*. Open datasets belonging to other database components within the default session remain open.

Identifying the database

AliasName and *DriverName* are mutually exclusive properties that identify the database server to which the *TDatabase* component connects.

- *AliasName* specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

  You create and edit BDE aliases using the Database Explorer or the BDE Administration utility. For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

- *DriverName* is the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local BDE alias for a database component using the *DatabaseName* property. If you specify *DriverName*, any value already assigned to *AliasName* is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in *AliasName*.

DatabaseName lets you provide your own name for a database connection. The name you supply is in addition to *AliasName* or *DriverName*, and is local to your application. *DatabaseName* can be a BDE alias, or, for Paradox and dBASE files, a fully-qualified path name. Like *AliasName*, *DatabaseName* appears in subsequent drop-down lists for dataset components to let you link them to database components.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

You can enter a *DatabaseName* in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the *Alias* property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the *DriverName* property, or you can choose from existing driver names in the drop-down list.

Note The Database Properties editor also lets you view and set BDE connection parameters, and set the states of the *LoginPrompt* and *KeepConnection* properties. For information on connection parameters, see “Setting BDE alias parameters” below. For information on *LoginPrompt*, see “Controlling server login” on page 17-4. For information on *KeepConnection* see “Opening a connection using TDatabase” on page 20-15.

Setting BDE alias parameters

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer or BDE Administration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the Object Inspector to invoke the String List editor.
- Double-click a database component in a data module or form to invoke the Database Properties editor.

All of these methods edit the *Params* property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

At runtime, an application can set alias parameters only by editing the *Params* property directly. For more information about parameters specific to using SQL Links drivers with the BDE, see your online SQL Links help file.

Opening a connection using TDatabase

As with all database connection components, to connect to a database using *TDatabase*, you set the *Connected* property to *True* or call the *Open* method. This process is described in "Connecting to a database server" on page 17-3. Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, the connection is dropped unless the database component's *KeepConnection* property is *True*.

When you connect to a remote database server from an application, the application uses the BDE and the Borland SQL Links driver to establish the connection. (The BDE can also communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the *Params* property of a database component. For information about SQL Links parameters, see the online *SQL Links User's Guide*. To edit the *Params* property, see "Setting BDE alias parameters" on page 20-14.

Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver's configuration options. In most cases, network protocol configuration is handled using a server's client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- Are the DLLs for your connection and database drivers in the search path?
- If you are using TCP/IP:
  - Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?
  - Is the server's IP address registered in the client's HOSTS file?

- Is the Domain Name Services (DNS) properly configured?
- Can you ping the server?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

Using ODBC

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- The BDE Administration utility.

To set up a BDE alias for an ODBC driver connection, use the BDE Administration utility. For more information, see the BDE Administration utility's online help file.

Using database components in data modules

You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the *HandleShared* property of the database component to *True* to prevent global name space conflicts.

Managing database sessions

An BDE-based application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application.

All BDE-based database applications automatically include a default session component, named *Session*, that encapsulates the default BDE session. When database components are added to the application, they are automatically associated with the default session (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are implicit (created by the session at runtime when you open a dataset that is not associated with a database component you create) or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

To use the default session, you need write no code unless your application must

- Explicitly activate or deactivate a session, enabling or disabling the session's databases' ability to open.
- Modify the properties of the session, such as specifying default properties for implicitly generated database components.

- Execute a session's methods, such as managing database connections (for example opening and closing database connections in response to user actions).
- Respond to session events, such as when the application attempts to access a password-protected Paradox or dBASE table.
- Set Paradox directory locations such as the *NetFileDir* property to access Paradox tables on a network and the *PrivateDir* property to a local hard drive to speed performance.
- Manage the BDE aliases that describe possible database connection configurations for databases and datasets that use the session.

Whether you add database components to an application at design time or create them dynamically at runtime, they are automatically associated with the default session unless you specifically assign them to a different session. If you open a dataset that is not associated with a database component, Delphi automatically

- Creates a database component for it at runtime.
- Associates the database component with the default session.
- Initializes some of the database component's key properties based on the default session's properties. Among the most important of these properties is *KeepConnections*, which determines when database connections are maintained or dropped by an application.

The default session provides a widely applicable set of defaults that can be used as is by most applications. You need only associate a database component with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions, where each thread has its own session.

Applications can create additional session components as needed. BDE-based database applications automatically include a session list component, named *Sessions*, that you can use to manage all of your session components. For more information about managing multiple sessions see, "Managing multiple sessions" on page 20-28.

You can safely place session components in data modules. If you put a data module that contains one or more session components into the Object Repository, however, make sure to set the *AutoSessionName* property to *True* to avoid namespace conflicts when users inherit from it.

Activating a session

Active is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is associated with a session is opened and there are currently no other open databases or datasets). Setting *Active* to *True* triggers a session's *OnStartup* event, registers the paradox directory locations with the BDE, and registers the *ConfigMode* property, which determines what BDE aliases are available within the session. You can write an *OnStartup* event handler to initialize the *NetFileDir*, *PrivateDir*, and *ConfigMode* properties before they are registered with the BDE, or to perform other specific session start-up activities. For information about the *NetFileDir* and *PrivateDir* properties, see "Specifying Paradox directory locations" on page 20-24. For information about ConfigMode, see "Working with BDE aliases" on page 20-24.

Once a session is active, you can open its database connections by calling the *OpenDatabase* method.

For session components you place in a data module or form, setting *Active* to *False* when there are open databases or datasets closes them. At runtime, closing databases and datasets may trigger events associated with them.

Note You cannot set *Active* to *False* for the default session at design time. While you can close the default session at runtime, it is not recommended.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following single line of code closes all open databases and datasets for a session:

```
Session1.Close;
```

This code sets Session1's *Active* property to *False*. When a session's *Active* property is *False*, any subsequent attempt by the application to open a database or dataset resets *Active* to *True* and calls the session's *OnStartup* event handler if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
Session1.Open;
```

Note If a session is active you can also open and close individual database connections. For more information, see "Closing database connections" on page 20-19.

Specifying default database connection behavior

KeepConnections provides the default value for the *KeepConnection* property of implicit database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed.

Note Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a database component always overrides the *KeepConnections* property of the session. For more information about controlling individual database connections within a session, see "Managing database connections" on page 20-19.

KeepConnections should be set to *True* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or reestablishes a connection, it incurs the overhead of attaching and detaching the database.

Note Even when *KeepConnections* is *True* for a session, you can close and free inactive database connections for all implicit database components by calling the *DropConnections* method. For more information about *DropConnections*, see "Dropping inactive database connections" on page 20-20.

Managing database connections

You can use a session component to manage the database connections within it. The session component includes properties and methods you can use to

- Open database connections.
- Close database connections.
- Close and free all inactive temporary database connections.
- Locate specific database connections.
- Iterate through all open database connections.

Opening database connections

To open a database connection within a session, call the *OpenDatabase* method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:

```
var
  DBDemosDatabase: TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
```

OpenDatabase actives the session if it is not already active, and then checks if the specified database name matches the *DatabaseName* property of any database components for the session. If the name does not match an existing database component, *OpenDatabase* creates a temporary database component using the specified name. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server. Each call to *OpenDatabase* increments a reference count for the database by 1. As long as this reference count remains greater than 0, the database is open.

Closing database connections

To close an individual database connection, call the *CloseDatabase* method. When you call *CloseDatabase*, the reference count for the database, which is incremented when you call *OpenDatabase*, is decremented by 1. When the reference count for a database is 0, the database is closed. *CloseDatabase* takes one parameter, the database to close. If you opened the database using the *OpenDatabase* method, this parameter can be set to the return value of *OpenDatabase*.

```
Session.CloseDatabase(DBDemosDatabase);
```

If the specified database name is associated with a temporary (implicit) database component, and the session's *KeepConnections* property is *False*, the database component is freed, effectively closing the connection.

Note If *KeepConnections* is *False* temporary database components are closed and freed automatically when the last dataset associated with the database component is closed. An application can always call *CloseDatabase* prior to that time to force closure. To free temporary database components when *KeepConnections* is *True*, call the database component's *Close* method, and then call the session's *DropConnections* method.

Note Calling *CloseDatabase* for a persistent database component does not actually close the connection. To close the connection, call the database component's *Close* method directly.

There are two ways to close all database connections within the session:

- Set the *Active* property for the session to *False*.
- Call the *Close* method for the session.

When you set *Active* to *False*, Delphi automatically calls the *Close* method. *Close* disconnects from all active databases by freeing temporary database components and calling each persistent database component's *Close* method. Finally, *Close* sets the session's BDE handle to **nil**.

Dropping inactive database connections

If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the *DropConnections* method. For example, the following code frees all inactive, temporary database components for the default session:

```
Session.DropConnections;
```

Temporary database components for which one or more datasets are active are not dropped or freed by this call. To free these components, call *Close*.

Searching for a database connection

Use a session's *FindDatabase* method to determine whether a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, it can also be a fully-qualified path name.

FindDatabase returns the database component if it finds a match. Otherwise it returns **nil**.

The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

```
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
```

```
    if (DB = nil) then                          { database doesn't exist for session so,}
      DB := Session.OpenDatabase('DBDEMOS');    { create and open it}
    if Assigned(DB) and DB.Connected then begin
      DB.StartTransaction;
      ...
    end;
  end;
```

Iterating through a session's database components

You can use two session component properties, *Databases* and *DatabaseCount*, to cycle through all the active database components associated with a session.

Databases is an array of all currently active database components associated with a session. *DatabaseCount* is the number of databases in that array. As connections are opened or closed during a session's life-span, the values of *Databases* and *DatabaseCount* change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
end;
```

Working with password-protected Paradox and dBASE tables

A session component can store passwords for password-protected Paradox and dBASE tables. Once you add a password to the session, your application can open tables protected by that password. Once you remove the password from the session, your application can't open tables that use the password until you add it again.

Using the AddPassword method

The *AddPassword* method provides an optional way for an application to provide a password for a session prior to opening an encrypted Paradox or dBASE table that requires a password for access. If you do not add the password to the session, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

AddPassword takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords (one at a time) to access tables protected with different passwords.

```
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrd);
  try
    Table1.Open;
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;
```

Note Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form.

The Add button of the *PasswordDialog* function dialog has the same effect as the *AddPassword* method.

```
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;
```

Using the RemovePassword and RemoveAllPasswords methods

RemovePassword deletes a previously added password from memory. *RemovePassword* takes one parameter, a string containing the password to delete.

```
Session.RemovePassword(‘secret’);
```

RemoveAllPasswords deletes all previously added passwords from memory.

```
Session.RemoveAllPasswords;
```

Using the GetPassword method and OnPassword event

The *OnPassword* event allows you to control how your application supplies passwords for Paradox and dBASE tables when they are required. Provide a handler for the *OnPassword* event if you want to override the default password handling behavior. If you do not provide a handler, Delphi presents a default dialog for entering a password and no special behavior is provided—the table open attempt either succeeds or an exception is raised.

If you provide a handler for the *OnPassword* event, do two things in the event handler: call the *AddPassword* method and set the event handler’s *Continue* parameter to *True*. The *AddPassword* method passes a string to the session to be used as a password for the table. The *Continue* parameter indicates to Delphi that no further password prompting need be done for this table open attempt. The default value for *Continue* is *False*, and so requires explicitly setting it to *True*. If *Continue* is *False* after the event handler has finished executing, an *OnPassword* event fires again—even if a valid password has been passed using *AddPassword*. If *Continue* is *True* after execution of the event handler and the string passed with *AddPassword* is not the valid password, the table open attempt fails and an exception is raised.

OnPassword can be triggered by two circumstances. The first is an attempt to open a password-protected table (dBASE or Paradox) when a valid password has not already been supplied to the session. (If a valid password for that table has already been supplied, the *OnPassword* event does not occur.)

The other circumstance is a call to the *GetPassword* method. *GetPassword* either generates an *OnPassword* event, or, if the session does not have an *OnPassword* event handler, displays a default password dialog. It returns *Tru*e if the *OnPassword* event handler or default dialog added a password to the session, and *False* if no entry at all was made.

In the following example, the *Password* method is designated as the *OnPassword* event handler for the default session by assigning it to the global *Session* object's *OnPassword* property.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
```

In the *Password* method, the *InputBox* function prompts the user for a password. The *AddPassword* method then programmatically supplies the password entered in the dialog to the session.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Continue := (Passwrd > '');
  Session.AddPassword(Passwrd);
end;
```

The *OnPassword* event (and thus the *Password* event handler) is triggered by an attempt to open a password-protected table, as demonstrated below. Even though the user is prompted for a password in the handler for the *OnPassword* event, the table open attempt can still fail if they enter an invalid password or something else goes wrong.

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
  CRLF = #13 + #10;
begin
  try
    Table1.Open;                              { this line triggers the OnPassword event }
  except
    on E:Exception do begin                            { exception if cannot open table }
      ShowMessage('Error!' + CRLF +             { display error explaining what happened }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate;                                      { end the application }
    end;
  end;
end;
```

Specifying Paradox directory locations

Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables.

NetFileDir specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server). Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so be sure to validate the new value.

At design time, you can specify a value for *NetFileDir* in the Object Inspector. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

Note *NetFileDir* can only be changed when an application does not have any open Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

PrivateDir specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements. If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you can improve application performance at runtime by setting *PrivateDir* to a user's local hard drive before opening the database.

Note Do not set *PrivateDir* at design time and then open the session in the IDE. Doing so generates a Directory is busy error when running your application from the IDE.

The following code changes the setting of the default session's *PrivateDir* property to a user's C:\TEMP directory:

```
Session.PrivateDir := 'C:\TEMP';
```

Important Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

Working with BDE aliases

Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables). A session can create, modify, and delete aliases during its lifetime.

The *AddAlias* method creates a new BDE alias for an SQL database server. *AddAlias* takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For example, the following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
```

```
    try
      with AliasParams do begin
        Add('OPEN MODE=READ');
        Add('USER NAME=TOMSTOPPARD');
        Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
      end;
      Session.AddAlias('CATS', 'INTRBASE', AliasParams);
      ...
    finally
      AliasParams.Free;
    end;
  end;
```

AddStandardAlias creates a new BDE alias for Paradox, dBASE, or ASCII tables. *AddStandardAlias* takes three string parameters: the name for the alias, the fully-qualified path to the Paradox or dBASE table to access, and the name of the default driver to use when attempting to open a table that does not have an extension. For example, the following statement uses *AddStandardAlias* to create a new alias for accessing a Paradox table:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

When you add an alias to a session, the BDE stores a copy of the alias in memory, where it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. *ConfigMode* is a set that describes which types of aliases can be used by the databases in the session. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent*, *cfmSession*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session (*cfmSession*), all aliases in the BDE configuration file on a user's system (*cfmPersistent*), and all aliases that the BDE maintains in memory (*cfmVirtual*). You can change *ConfigMode* to restrict what BDE aliases the databases in a session can use. For example, setting *ConfigMode* to *cfmSession* restricts a session's view of aliases to those created within the session. All other aliases in the BDE configuration file and in memory are not available.

To make a newly created alias available to all sessions and to other applications, use the session's *SaveConfigFile* method. *SaveConfigFile* writes aliases in memory to the BDE configuration file where they can be read and used by other BDE-enabled applications.

After you create an alias, you can make changes to its parameters by calling *ModifyAlias*. *ModifyAlias* takes two parameters: the name of the alias to modify and a string list containing the parameters to change and their values. For example, the following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

To delete an alias previously created in a session, call the *DeleteAlias* method. *DeleteAlias* takes one parameter, the name of the alias to delete. *DeleteAlias* makes an alias unavailable to the session.

Note *DeleteAlias* does not remove an alias from the BDE configuration file if the alias was written to the file by a previous call to *SaveConfigFile*. To remove the alias from the configuration file after calling *DeleteAlias*, call *SaveConfigFile* again.

Session components provide five methods for retrieving information about a BDE aliases, including parameter information and driver information. They are:

- *GetAliasNames*, to list the aliases to which a session has access.
- *GetAliasParams*, to list the parameters for a specified alias.
- *GetAliasDriverName*, to return the name of the BDE driver used by the alias.
- *GetDriverNames*, to return a list of all BDE drivers available to the session.
- *GetDriverParams*, to return driver parameters for a specified driver.

For more information about using a session's informational methods, see "Retrieving information about a session" below. For more information about BDE aliases and the SQL Links drivers with which they work, see the BDE online help, BDE32.HLP.

Retrieving information about a session

You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method retrieves the names of tables associated with a specific database component used by the session. Table 20.4 summarizes the informational methods to a session component:

Table 20.4 Database-related informational methods for session components

Method	Purpose
GetAliasDriverName	Retrieves the BDE driver for a specified alias of a database.
GetAliasNames	Retrieves the list of BDE aliases for a database.
GetAliasParams	Retrieves the list of parameters for a specified BDE alias of a database.
GetConfigParams	Retrieves configuration information from the BDE configuration file.
GetDatabaseNames	Retrieves the list of BDE aliases and the names of any *TDatabase* components currently in use.
GetDriverNames	Retrieves the names of all currently installed BDE drivers.
GetDriverParams	Retrieves the list of parameters for a specified BDE driver.
GetStoredProcNames	Retrieves the names of all stored procedures for a specified database.
GetTableNames	Retrieves the names of all tables matching a specified pattern for a specified database.
GetFieldNames	Retrieves the names of all fields in a specified table in a specified database.

Except for *GetAliasDriverName*, these methods return a set of values into a string list declared and maintained by your application. (*GetAliasDriverName* returns a single string, the name of the current BDE driver for a particular database component used by the session.)

For example, the following code retrieves the names of all database components and aliases known to the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

Creating additional sessions

You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the Object Inspector, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime.

Note Creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.

To enable dynamic creation of a session component at runtime, follow these steps:

1. Declare a *TSession* variable.
2. Instantiate a new session by calling the *Create* method. The constructor sets up an empty list of database components for the session, sets the *KeepConnections* property to *True*, and adds the session to the list of sessions maintained by the application's session list component.
3. Set the *SessionName* property for the new session to a unique name. This property is used to associate database components with the session. For more information about the *SessionName* property, see "Naming a session" on page 20-28.
4. Activate the session and optionally adjust its properties.

You can also create and open sessions using the *OpenSession* method of *TSessionList*. Using *OpenSession* is safer than calling *Create*, because *OpenSession* only creates a session if it does not already exist. For information about *OpenSession*, see "Managing multiple sessions" on page 20-28.

The following code creates a new session component, assigns it a name, and opens the session for database operations that follow (not shown here). After use, it is destroyed with a call to the *Free* method.

Note Never delete the default session.

```
var
  SecondSession: TSession;
begin
  SecondSession := TSession.Create(Form1);
  with SecondSession do
    try
```

```
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
end;
```

Naming a session

A session’s *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is “Default,” For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the *SessionName* of a session component you create.

The following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to “InterBaseSession,” activate the session, and associate an existing database component *Database1* with that session:

```
var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

Managing multiple sessions

If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The BDE page on the Component palette contains a session component that you can place in a data module or on a form at design time.

Important When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session’s *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the *OpenSession* method of the global *Sessions* object at runtime.

OpenSession requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

Sessions is a variable of type *TSessionList* that is automatically instantiated for BDE-based database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. Table 20.5 summarizes the properties and methods of the *TSessionList* component:

Table 20.5 TSessionList properties and methods

Property or Method	Purpose
Count	Returns the number of sessions, both active and inactive, in the session list.
FindSession	Searches for a session with a specified name and returns a pointer to it, or **nil** if there is no session with the specified name. If passed a blank session name, *FindSession* returns a pointer to the default session, *Session*.
GetSessionNames	Populates a string list with the names of all currently instantiated session components. This procedure always adds at least one string, “Default” for the default session.
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value.

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

If you are replicating a data module that contains a session in a multi-threaded application, where each thread contains its own copy of the data module, you can use the *AutoSessionName* property to make sure that all datasets in the data module use the correct session. Setting *AutoSessionName* to *True* causes the session to generate its own unique name dynamically when it is created at runtime. It then assigns this name to every dataset in the data module, overriding any explicitly set session names. This ensures that each thread has its own session, and each dataset uses the session in its own data module.

Using transactions with the BDE

By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly.

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the database component to control transactions. The main advantage to using the methods and properties of a database component is that it provides a clean, portable application that is not dependent on a particular database or server. This type of transaction control is supported by all database connection components, and described in “Managing transactions” on page 17-5
- Use passthrough SQL in a query component to pass SQL statements directly to remote SQL or ODBC servers. The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server’s transaction management model, see your database server documentation. For more information about using passthrough SQL, see “Using passthrough SQL” below.

When working with local databases, you can only use the database component to create explicit transactions (local databases do not support passthrough SQL). However, there are limitations to using local transactions. For more information on using local transactions, see “Using local transactions” on page 20-31.

Note You can minimize the number of transactions you need by caching updates. For more information about cached updates, see “Using a client dataset to cache updates” on page 23-15 and “Using the BDE to cache updates” on page 20-32.

Using passthrough SQL

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To use passthrough SQL to control a transaction, you must

- Install the proper SQL Links drivers. If you chose the “Typical” installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol. See your network administrator for more information.
- Have access to a database on a remote server.
- Set SQLPASSTHRU MODE to NOT SHARED using the SQL Explorer. SQLPASSTHRU MODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHRU MODE is set to SHARED AUTOCOMMIT. However, you can’t share database connections when using transaction control statements. For more information about SQLPASSTHRU modes, see the help file for the BDE Administration utility.

Note When SQLPASSTHRU MODE is NOT SHARED, you must use separate database components for datasets that pass SQL transaction statements to the server and datasets that do not.

Using local transactions

The BDE supports local transactions against Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

Note When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Local transactions are more limited than transactions against SQL servers or ODBC drivers. In particular, the following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- Transactions cannot be run against temporary tables.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Only a limited number of records can be locked and modified. With Paradox tables, you are limited to 255 records. With dBASE the limit is 100.

- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
  - Several tables are open.
  - The cursor is closed on a table to which no changes were made.

Using the BDE to cache updates

The recommended approach for caching updates is to use a client dataset (*TBDEClientDataSet*) or to connect the BDE-dataset to a client dataset using a dataset provider. The advantages of using a client dataset are discussed in "Using a client dataset to cache updates" on page 23-15.

For simple cases, however, you may choose to use the BDE to cache updates instead. BDE-enabled datasets and *TDatabase* components provide built-in properties, methods, and events for handling cached updates. Most of these correspond directly to the properties, methods, and events that you use with client datasets and dataset providers when using a client dataset to cache updates. The following table lists these properties, events, and methods and the corresponding properties, methods and events on *TBDEClientDataSet*:

Table 20.6 Properties, methods, and events for cached updates

On BDE-enabled datasets (or TDatabase)	On TBDEClientDataSet	Purpose
CachedUpdates	Not needed for client datasets, which always cache updates.	Determines whether cached updates are in effect for the dataset.
UpdateObject	Use a *BeforeUpdateRecord* event handler, or, if using *TClientDataSet*, use the *UpdateObject* property on the BDE-enabled source dataset.	Specifies the update object for updating read-only datasets.
UpdatesPending	*ChangeCount*	Indicates whether the local cache contains updated records that need to be applied to the database.
UpdateRecordTypes	*StatusFilter*	Indicates the kind of updated records to make visible when applying cached updates.
UpdateStatus	*UpdateStatus*	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError	*OnReconcileError*	An event for handling update errors on a record-by-record basis.
OnUpdateRecord	*BeforeUpdateRecord*	An event for processing updates on a record-by-record basis.
ApplyUpdates *ApplyUpdates* (database)	*ApplyUpdates*	Applies records in the local cache to the database.

Table 20.6 Properties, methods, and events for cached updates (continued)

On BDE-enabled datasets (or TDatabase)	On TBDEClientDataSet	Purpose
CancelUpdates	*CancelUpdates*	Removes all pending updates from the local cache without applying them.
CommitUpdates	*Reconcile*	Clears the update cache following successful application of updates.
FetchAll	*GetNextPacket* (and *PacketRecords*)	Copies database records to the local cache for editing and updating.
RevertRecord	*RevertRecord*	Undoes updates to the current record if updates are not yet applied.

For an overview of the cached update process, see "Overview of using cached updates" on page 23-16.

Note Even if you are using a client dataset to cache updates, you may want to read the section about update objects on page 20-39. You can use update objects in the *BeforeUpdateRecord* event handler of *TBDEClientDataSet* or *TDataSetProvider* to apply updates from stored procedures or multi-table queries.

Enabling BDE-based cached updates

To use the BDE for cached updates, the BDE-enabled dataset must indicate that it should cache updates. This is specified by setting the *CachedUpdates* property to *True*. When you enable cached updates, a copy of all records is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the application applies those changes to the database server. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

The dataset caches all updates until you set *CachedUpdates* to *False*. Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory. Canceling the updates by calling *CancelUpdates* removes all the changes currently in the cache, but does not stop the dataset from caching any subsequent changes.

Note If you disable cached updates by setting *CachedUpdates* to *False*, any pending changes that you have not yet applied are discarded without notification. To prevent losing changes, test the *UpdatesPending* property before disabling cached updates.

Applying BDE-based cached updates

Applying updates is a two-phase process that should occur in the context of a database component's transaction so that your application can recover gracefully from errors. For information about transaction handling with database components, see "Managing transactions" on page 17-5.

When applying updates under database transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.
3. The transaction is committed if writes are successful or rolled back if they are not:

   If the database write is successful:

   - Database changes are committed, ending the database transaction.
   - Cached updates are committed, clearing the internal cache buffer (phase 2).

   If the database write is unsuccessful:

   - Database changes are rolled back, ending the database transaction.
   - Cached updates are not committed, remaining intact in the internal cache.

For information about creating and using an *OnUpdateRecord* event handler, see "Creating an OnUpdateRecord event handler" on page 20-36. For information about handling update errors that occur when applying cached updates, see "Handling cached update errors" on page 20-37.

Note Applying cached updates is particularly tricky when you are working with multiple datasets linked in a master/detail relationship because the order in which you apply updates to each dataset is significant. Usually, you must update master tables before detail tables, except when handling deleted records, where this order must be reversed. Because of this difficulty, it is strongly recommended that you use client datasets when caching updates in a master/detail form. Client datasets automatically handle all ordering issues with master/detail relationships.

There are two ways to apply BDE-based updates:

- You can apply updates using a database component by calling its *ApplyUpdates* method. This method is the simplest approach, because the database handles all details of managing a transaction for the update process and of clearing the dataset's cache when updating is complete.
- You can apply updates for a single dataset by calling the dataset's *ApplyUpdates* and *CommitUpdates* methods. When applying updates at the dataset level you must explicitly code the transaction that wraps the update process as well as explicitly call *CommitUpdates* to commit updates from the cache.

Important To apply updates from a stored procedure or an SQL query that does not return a live result set, you must use *TUpdateSQL* to specify how to perform updates. For updates to joins (queries involving two or more tables), you must provide one *TUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. See "Using update objects to update a dataset" on page 20-39 for details.

Applying cached updates using a database

To apply cached updates to one or more datasets in the context of a database connection, call the database component's *ApplyUpdates* method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
```

The above sequence writes cached updates to the database in the context of an automatically-generated transaction. If successful, it commits the transaction and then commits the cached updates. If unsuccessful, it rolls back the transaction and leaves the update cache unchanged. In this latter case, you should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see "Handling cached update errors" on page 20-37.

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TDBDataSet*. For example, the following code applies updates for two queries:

```
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
  Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

Applying cached updates with dataset component methods

You can apply updates for individual BDE-enabled datasets directly using the dataset's *ApplyUpdates* and *CommitUpdates* methods. Each of these methods encapsulate one phase of the update process:

1. *ApplyUpdates* writes cached changes to a database (phase 1).
2. *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset:

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  Database1.StartTransaction;
  try
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates;              { try to write the updates to the database }
    Database1.Commit;                                   { on success, commit the changes }
  except
    Database1.Rollback;                                   { on failure, undo any changes }
```

```
      raise;              { raise the exception again to prevent a call to CommitUpdates }
    end;
    CustomerQuery.CommitUpdates;                    { on success, clear the internal cache }
  end;
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The *raise* statement inside the try...except block re*raises* the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

Creating an OnUpdateRecord event handler

When a BDE-enabled dataset applies its cached updates, it iterates through the changes recorded in its cache, attempting to apply them to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, special parameter substitution, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update that needs to be performed for the current record. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. If you are using an update object, you need to pass this parameter to the update object when applying the update. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update.

The *UpdateAction* parameter indicates whether you applied the update. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. If your event handler successfully applies the update, change this parameter to *uaApplied* before exiting. If you decide not to update the current record, change the value to *uaSkip* to preserve unapplied changes in the cache. If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted and an exception is raised. You can suppress the error message (raising a silent exception) by changing *UpdateAction* to *uaAbort*.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. *OldValue* gives the original field value that was fetched from the database. It can be useful in locating the database record to update. *NewValue* is the edited value in the update you are trying to apply.

Important An *OnUpdateRecord* event handler, like an *OnUpdateError* or *OnCalcFields* event handler, should never call any methods that change the current record in a dataset.

The following example illustrates how to use these parameters and properties. It uses a *TTable* component named *UpdateTable* to apply updates. In practice, it is easier to use an update object, but using a table illustrates the possibilities more clearly.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
        UpdateAction := uaApplied;
end;
```

Handling cached update errors

The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports any errors. The dataset component's *OnUpdateError* event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Here is the skeleton code for an *OnUpdateError* event handler:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
```

DataSet references the dataset to which updates are applied. You can use this dataset to access new and old values during error handling. The original values for fields in each record are stored in a read-only *TField* property called *OldValue*. Changed values are stored in the analogous *TField* property *NewValue*. These values provide the only way to inspect and change update values in the event handler.

Warning Do not call any dataset methods that change the current record (such as *Next* and *Prior*). Doing so causes the event handler to enter an endless loop.

The *E* parameter is usually of type *EDBEngineError*. From this exception type, you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
ErrorLabel.Caption := E.Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it.

The *UpdateKind* parameter describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

Table 20.7 UpdateKind values

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting a new record caused an error.
ukDelete	Deleting an existing record caused an error.

UpdateAction tells the BDE how to proceed with the update process when your event handler exits. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler:

- If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.
- When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.
- Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

The following code shows an *OnUpdateError* event handler that checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip                      { key violation, just skip this record }
    else
      UpdateAction := uaAbort;              { don't know what's wrong, abort the update }
  end;
```

Note If an error occurs during the application of cached updates, an exception is *raised* and an error message displayed. Unless the *ApplyUpdates* is called from within a try...except construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

Using update objects to update a dataset

When the BDE-enabled dataset represents a stored procedure or a query that is not "live", it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates. Whether you are using the BDE or a client dataset to cache updates, you can handle these problem datasets by using an update object:

1. If you are using a client dataset, use an external provider component with *TClientDataSet* rather than *TBDEClientDataSet*. This is so you can set the *UpdateObject* property of the BDE-enabled source dataset (step 3).
2. Add a *TUpdateSQL* component to the same data module as the BDE-enabled dataset.
3. Set the BDE-enabled dataset component's *UpdateObject* property to the *TUpdateSQL* component in the data module.
4. Specify the SQL statements needed to perform updates using the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. You can use the Update SQL editor to help you compose these statements.
5. Close the dataset.
6. Set the dataset component's *CachedUpdates* property to *True* or link the dataset to the client dataset using a dataset provider.
7. Reopen the dataset.

Note Sometimes, you need to use multiple update objects. For example, when updating a multi-table join or a stored procedure that represents data from multiple datasets, you must provide one *TUpdateSQL* object for each table you want to update. When using multiple update objects, you can't simply associate the update object with the dataset by setting the *UpdateObject* property. Instead, you must manually call the update object from an *OnUpdateRecord* event handler (when using the BDE to cache updates) or a *BeforeUpdateRecord* event handler (when using a client dataset).

The update object actually encapsulates three *TQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties for which you supply SQL statements:

- *ModifySQL* specifies the UPDATE statement.
- *InsertSQL* specifies the INSERT statement.
- *DeleteSQL* specifies the DELETE statement.

At runtime, when the update component is used to apply updates, it:

1. Selects an SQL statement to execute based on whether the current record is modified, inserted, or deleted.
2. Provides parameter values to the SQL statement.
3. Prepares and executes the SQL statement to perform the specified update.

Creating SQL statements for update components

To update a record in an associated dataset, an update object uses one of three SQL statements. Each update object can only update a single table, so the object's update statements must each reference the same base table.

The three SQL statements delete, insert, and modify records cached for update. You must provide these statements as update object's *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. You can provide these values at design time or at runtime. For example, the following code specifies a value for the *DeleteSQL* property at runtime:

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

At design time, you can use the Update SQL editor to help you compose the SQL statements that apply updates.

Update objects provide automatic parameter binding for parameters that reference the dataset's original and updated field values. Typically, therefore, you insert parameters with specially formatted names when you compose the SQL statements. For information on using these parameters, see "Understanding parameter substitution in update SQL statements" on page 20-41.

Using the Update SQL editor

To create the SQL statements for an update component,

1. Using the Object Inspector, select the name of the update object from the drop-down list for the dataset's *UpdateObject* property. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.
2. Right-click the update object and select UpdateSQL Editor from the context menu. This displays the Update SQL editor. The editor creates SQL statements for the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. For Paradox, dBASE, and FoxPro the columns you specify here must correspond to an existing index, but this is not a requirement for remote SQL databases. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. In most cases you will want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Important Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather then using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

Understanding parameter substitution in update SQL statements

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string

"OLD_", then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the ":OLD_FieldName" syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer's last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with "Smith" as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

Note If you create SQL statements that contain parameters that do not refer the edited or original field values, the update object does not know how to bind their values. You can, however, do this manually, using the update object's *Query* property. See "Using an update component's Query property" on page 20-46 for details.

Composing update SQL statements

At design time, you can use the Update SQL editor to write the SQL statements for the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. If you do not use the Update SQL editor, or if you want to modify the generated statements, you should keep in mind the following guidelines when writing statements to delete, insert, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with "OLD_", the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some table types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with "OLD_". In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

Using multiple update objects

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with a dataset by setting its *DataSet* property to the name of the dataset.

Tip When using multiple update objects, you can use *TBDEClientDataSet* instead of *TClientDataSet* with an external provider. This is because you do not need to set the source dataset's *UpdateObject* property.

The *DataSet* property for update objects is not available at design time in the Object Inspector. You can only set this property at runtime.

```
UpdateSQL1.DataSet := Query1;
```

The update object uses this dataset to obtain original and updated field values for parameter substitution and, if it is a BDE-enabled dataset, to identify the session and database to use when applying the updates. So that parameter substitution will work correctly, the update object's *DataSet* property must be the dataset that contains the updated field values. When using the BDE-enabled dataset to cache updates, this is the BDE-enabled dataset itself. When using a client dataset, this is a client dataset that is provided as a parameter to the *BeforeUpdateRecord* event handler.

When the update object has not been assigned to the dataset's *UpdateObject* property, its SQL statements are not automatically executed when you call *ApplyUpdates*. To update records, you must manually call the update object from an *OnUpdateRecord* event handler (when using the BDE to cache updates) or a *BeforeUpdateRecord* event handler (when using a client dataset). In the event handler, the minimum actions you need to take are

- If you are using a client dataset to cache updates, you must be sure that the updates object's *DatabaseName* and *SessionName* properties are set to the *DatabaseName* and *SessionName* properties of the source dataset.
- The event handler must call the update object's *ExecSQL* or *Apply* method. This invokes the update object for each record that requires updating. For more information about executing update statements, see "Executing the SQL statements" below.
- Set the event handler's *UpdateAction* parameter to *uaApplied* (*OnUpdateRecord*) or the *Applied* parameter to *True* (*BeforeUpdateRecord*).

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

Warning If you call an update object's *ExecSQL* or *Apply* method in an *OnUpdateRecord* event handler, be sure that you do not set the dataset's *UpdateObject* property to that update object. Otherwise, this will result in a second attempt to apply each record's update.

Executing the SQL statements

When you use multiple update objects, you do not associate the update objects with a dataset by setting its *UpdateObject* property. As a result, the appropriate statements are not automatically executed when you apply updates. Instead, you must explicitly invoke the update object in code.

There are two ways to invoke the update object. Which way you choose depends on whether the SQL statement uses parameters to represent field values:

- If the SQL statement to execute uses parameters, call the *Apply* method.

- If the SQL statement to execute does not use parameters, it is more efficient to call the *ExecSQL* method.

Note If the SQL statement uses parameters other than the built-in types (for the original and updated field values), you must manually supply parameter values instead of relying on the parameter substitution provided by the *Apply* method. See "Using an update component's Query property" on page 20-46 for information on manually providing parameter values.

For information about the default parameter substitution for parameters in an update object's SQL statements, see "Understanding parameter substitution in update SQL statements" on page 20-41.

Calling the Apply method

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Initial and edited field values for the record are bound to parameters in the appropriate SQL statement.
2. The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event or from a provider's *BeforeUpdateRecord* event handler.

Warning If you use the dataset's *UpdateObject* property to associate dataset and update object, *Apply* is called automatically. In that case, do not call *Apply* in an *OnUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

OnUpdateRecord event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *Apply* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DataSet := DeltaDS;
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    Apply(UpdateKind);
    Applied := True;
  end;
end;
```

Calling the ExecSQL method

The *ExecSQL* method for an update component manually applies updates for the current record. Unlike the *Apply* method, *ExecSQL* does not bind parameters in the SQL statement before executing it. The *ExecSQL* method is most often called from within a handler for the *OnUpdateRecord* event (when using the BDE) or the *BeforeUpdateRecord* event (when using a client dataset).

Because *ExecSQL* does not bind parameter values, it is used primarily when the update object's SQL statements do not include parameters. You can use *Apply* instead, even when there are no parameters, but *ExecSQL* is more efficient because it does not check for parameters.

If the SQL statements include parameters, you can still call *ExecSQL*, but only after explicitly binding parameters. If you are using the BDE to cache updates, you can explicitly bind parameters by setting the update object's *DataSet* property and then calling its *SetParams* method. When using a client dataset to cache updates, you must supply parameters to the underlying query object maintained by *TUpdateSQL*. For information on how to do this, see "Using an update component's Query property" on page 20-46.

Warning If you use the dataset's *UpdateObject* property to associate dataset and update object, *ExecSQL* is called automatically. In that case, do not call *ExecSQL* in an *OnUpdateRecord* or *BeforeUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

OnUpdateRecord and *BeforeUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *ExecSQL* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
         DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ExecSQL(UpdateKind);
    Applied := True;
  end;
end;
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Using an update component's Query property

The *Query* property of an update component provides access to the query components that implement its *DeleteSQL*, *InsertSQL*, and *ModifySQL* statements. In most applications, there is no need to access these query components directly: you can use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to specify the statements these queries execute, and execute them by calling the update object's *Apply* or *ExecSQL* method. There are times, however, when you may need to directly manipulate the query component. In particular, the *Query* property is useful when you want to supply your own values for parameters in the SQL statements rather than relying on the update object's automatic parameter binding to old and new field values.

Note The *Query* property is only accessible at runtime.

The *Query* property is indexed on a *TUpdateKind* value:

- Using an index of *ukModify* accesses the query that updates existing records.
- Using an index of *ukInsert* accesses the query that inserts new records.
- Using an index of *ukDelete* accesses the query that deletes records.

The following shows how to use the *Query* property to supply parameter values that can't be bound automatically:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS; { required for the automatic parameter substitution }
  with UpdateSQL1.Query[UpdateKind] do
  begin
    { Make sure the query has the correct DatabaseName and SessionName }
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ParamByName('TimeOfUpdate').Value = Now;
  end;
  UpdateSQL1.Apply(UpdateKind); { now perform automatic substitutions and execute }
  Applied := True;
end;
```

Using TBatchMove

TBatchMove encapsulates Borland Database Engine (BDE) features that let you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

Creating a batch move component

To create a batch move component:

1. Place a table or query component for the dataset from which you want to import records (called the *Source* dataset) on a form or in a data module.
2. Place the dataset to which to move records (called the *Destination* dataset) on the form or data module.

3 Place a *TBatchMove* component from the BDE page of the Component palette in the data module or form, and set its *Name* property to a unique value appropriate to your application.

4 Set the *Source* property of the batch move component to the name of the table from which to copy, append, or update records. You can select tables from the drop-down list of available dataset components.

5 Set the *Destination* property to the dataset to create, append to, or update. You can select a destination table from the drop-down list of available dataset components.

- If you are appending, updating, or deleting, *Destination* must represent an existing database table.
- If you are copying a table and *Destination* represents an existing table, executing the batch move overwrites all of the current data in the destination table.
- If you are creating an entirely new table by copying an existing table, the resulting table has the name specified in the *Name* property of the table component to which you are copying. The resulting table type will be of a structure appropriate to the server specified by the *DatabaseName* property.

6 Set the *Mode* property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For information about these modes, see “Specifying a batch move mode” on page 20-49.

7 Optionally set the *Transliterate* property. If *Transliterate* is *True* (the default), character data is translated from the *Source* dataset’s character set to the *Destination* dataset’s character set as necessary.

8 Optionally set column mappings using the *Mappings* property. You need not set this property if you want batch move to match columns based on their position in the source and destination tables. For more information about mapping columns, see “Mapping data types” on page 20-50.

9 Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used. For more information about handling batch move errors, see “Handling batch move errors” on page 20-51.

Specifying a batch move mode

The *Mode* property specifies the operation a batch move component performs:

Table 20.8 Batch move modes

Property	Purpose
batAppend	Append records to the destination table.
batUpdate	Update records in the destination table with matching records from the source table. Updating is based on the current index of the destination table.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table.
batCopy	Create the destination table based on the structure of the source table. If the destination table already exists, it is dropped and recreated.
batDelete	Delete records in the destination table that match records in the source table.

Appending records

To append data, the destination dataset must represent an existing table. During the append operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary. If a conversion is not possible, an exception is thrown and the data is not appended.

Updating records

To update data, the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. During the update operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

Appending and updating records

To append and update data the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. Otherwise, data from the source dataset is appended to the destination dataset. During append and update operations, the BDE converts data to appropriate data types and sizes for the destination dataset, if necessary.

Copying datasets

To copy a source dataset, the destination dataset should not represent an exist table. If it does, the batch move operation overwrites the existing table with a copy of the source dataset.

If the source and destination datasets are maintained by different types of database engines, for example, Paradox and InterBase, the BDE creates a destination dataset with a structure as close as possible to that of the source dataset and automatically performs data type and size conversions as necessary.

Note *TBatchMove* does not copy metadata structures such as indexes, constraints, and stored procedures. You must recreate these metadata objects on your database server or through the SQL Explorer as appropriate.

Deleting records

To delete data in the destination dataset, it must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are deleted in the destination table.

Mapping data types

In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the *Mappings* property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a “best fit”. It trims character data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of ‘5’ to the corresponding integer value. Values that cannot be converted generate errors. For more information about errors, see “Handling batch move errors” on page 20-51.

When moving data between different table types, a batch move component translates data types as appropriate based on the dataset’s server types. See the BDE online help file for the latest tables of mappings among server types.

Note To batch move data to an SQL server database, you must have that database server and a version of Delphi with the appropriate SQL Link installed, or you can use ODBC if you have the proper third party ODBC drivers installed.

Executing a batch move

Use the *Execute* method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
BatchMoveAdd.Execute;
```

You can also execute a batch move at design time by right clicking the mouse on a batch move component and choosing Execute from the context menu.

The *MovedCount* property keeps track of the number of records that are moved when a batch move executes.

The *RecordCount* property specifies the maximum number of records to move. If *RecordCount* is zero, all records are moved, beginning with the first record in the source dataset. If *RecordCount* is a positive number, a maximum of *RecordCount* records are moved, beginning with the current record in the source dataset. If *RecordCount* is greater than the number of records between the current record in the source dataset and its last record, the batch move terminates when the end of the source dataset is reached. You can examine *MoveCount* to determine how many records were actually transferred.

Handling batch move errors

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. *TBatchMove* has a number of properties that report on and control error handling.

The *AbortOnProblem* property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems.

The *AbortOnKeyViol* property indicates whether to abort the operation when a Paradox key violation occurs.

The *ProblemCount* property indicates the number of records that could not be handled in the destination table without a loss of data. If *AbortOnProblem* is *True*, this number is one, since the operation is aborted when an error occurs.

The following properties enable a batch move component to create additional tables that document the batch move operation:

- *ChangedTableName*, if specified, creates a local Paradox table containing all records in the destination table that changed as a result of an update or delete operation.
- *KeyViolTableName,* if specified, creates a local Paradox table containing all records from the source table that caused a key violation when working with a Paradox table. If *AbortOnKeyViol* is *True*, this table will contain at most one entry since the operation is aborted on the first problem encountered.

- *ProblemTableName,* if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *True*, there is at most one record in this table since the operation is aborted on the first problem encountered.

Note If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table.

The Data Dictionary

When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a client/server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see “Creating attribute sets for field components” on page 19-12. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

A programming interface to the Data Dictionary is available in the drintf unit (located in the lib directory). This interface supplies the following methods:

Table 20.9 Data Dictionary interface

Routine	Use
DictionaryActive	Indicates if the data dictionary is active.
DictionaryDeactivate	Deactivates the data dictionary.
IsNullID	Indicates whether a given ID is a null ID
FindDatabaseID	Returns the ID for a database given its alias.
FindTableID	Returns the ID for a table in a specified database.
FindFieldID	Returns the ID for a field in a specified table.
FindAttrID	Returns the ID for a named attribute set.
GetAttrName	Returns the name an attribute set given its ID.
GetAttrNames	Executes a callback for each attribute set in the dictionary.

Table 20.9 Data Dictionary interface (continued)

Routine	Use
GetAttrID	Returns the ID of the attribute set for a specified field.
NewAttr	Creates a new attribute set from a field component.
UpdateAttr	Updates an attribute set to match the properties of a field.
CreateField	Creates a field component based on stored attributes.
UpdateField	Changes the properties of a field to match a specified attribute set.
AssociateAttr	Associates an attribute set with a given field ID.
UnassociateAttr	Removes an attribute set association for a field ID.
GetControlClass	Returns the control class for a specified attribute ID.
QualifyTableName	Returns a fully qualified table name (qualified by user name).
QualifyTableNameByName	Returns a fully qualified table name (qualified by user name).
HasConstraints	Indicates whether the dataset has constraints in the dictionary.
UpdateConstraints	Updates the imported constraints of a dataset.
UpdateDataset	Updates a dataset to the current settings and constraints in the dictionary.

Tools for working with the BDE

One advantage of using the BDE as a data access mechanism is the wealth of supporting utilities that ship with Delphi. These utilities include:

- **SQL Explorer** and **Database Explorer**: Delphi ships with one of these two applications, depending on which version you have purchased. Both Explorers enable you to
  - Examine existing database tables and structures. The SQL Explorer lets you examine and query remote SQL databases.
  - Populate tables with data
  - Create extended field attribute sets in the Data Dictionary or associate them with fields in your application.
  - Create and manage BDE aliases.

  SQL Explorer lets you do the following as well:
  - Create SQL objects such as stored procedures on remote database servers.
  - View the reconstructed text of SQL objects on remote database servers.
  - Run SQL scripts.
- **SQL Monitor**: SQL Monitor lets you watch all of the communication that passes between the remote database server and the BDE. You can filter the messages you want to watch, limiting them to only the categories of interest. SQL Monitor is most useful when debugging your application.

- **BDE Administration utility:** The BDE Administration utility lets you add new database drivers, configure the defaults for existing drivers, and create new BDE aliases.
- **Database Desktop**: If you are using Paradox or dBASE tables, Database Desktop lets you view and edit their data, create new tables, and restructure existing tables. Using Database Desktop affords you more control than using the methods of a *TTable* component (for example, it allows you to specify validity checks and language drivers). It provides the only mechanism for restructuring Paradox and dBASE tables other than making direct calls the BDE's API.

Chapter 21

Working with ADO components

The *ADOExpress* components provide data access through the ADO framework. ADO, (Microsoft ActiveX Data Objects) is a set of COM objects that access data through an OLE DB provider. The Delphi *ADOExpress* components encapsulate these ADO objects in the Delphi database architecture.

The ADO layer of an ADO-based application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are wrapped by the *TADOConnection*, *TADOCommand*, and ADO dataset components. The ADO framework includes other "helper" objects, like the Field and Properties objects, but these are typically not used directly in Delphi applications and are not wrapped by dedicated components.

This chapter presents the *ADOExpress* components and discusses the unique features they add to the common Delphi database architecture. Before reading about the features peculiar to the *ADOExpress* components, you should familiarize yourself with the common features of database connection components and datasets described in Chapter 17, "Connecting to databases" and Chapter 18, "Understanding datasets."

Overview of ADO components

The ADO page of the component palette hosts the *ADOExpress* components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most *ADOExpress* components have direct counterparts in the components available for other data access mechanisms: a database connection component (*TADOConnection*) and various types of datasets. In addition, *ADOExpress* includes *TADOCommand*, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The following table lists the ADO components.

Table 21.1 ADO components

Component	Use
TADOConnection	A database connection component that establishes a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
TADODataSet	The primary dataset for retrieving and operating on data; *TADODataSet* can retrieve data from a single or multiple tables; can connect directly to a data store or use a *TADOConnection* component.
TADOTable	A table-type dataset for retrieving and operating on a recordset produced by a single database table; *TADOTable* can connect directly to a data store or use a *TADOConnection* component.
TADOQuery	A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement; *TADOQuery* can also execute data definition language (DDL) SQL statements. It can connect directly to a data store or use a *TADOConnection* component
TADOStoredProc	A stored procedure-type dataset for executing stored procedures; *TADOStoredProc* executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a *TADOConnection* component.
TADOCommand	A simple component for executing commands (SQL statements that do not return result sets); *TADOCommand* can be used with a supporting dataset component, or retrieve a dataset from a table; It can connect directly to a data store or use a *TADOConnection* component.

Connecting to ADO data stores

Delphi ADO-based applications use Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider that connects to a data store and accesses its data. One of the items a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows.

An ADO provider represents one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must be installed on the client computer. OLE DB drivers for various database systems are supplied by the database vendor or by a third-party. If the application uses an SQL database, such as Microsoft SQL Server or Oracle, the client software for that database system must also be installed on the client computer. Client software is supplied by the database vendor and installed from the database systems CD (or disk).

To connect your application with the data store, use an ADO connection component (*TADOConnection*). Configure the ADO connection component to use one of the available ADO providers. Although *TADOConnection* is not strictly required, because ADO command and dataset components can establish connections directly using their *ConnectionString* property, you can use *TADOConnection* to share a single connection among several ADO components. This can reduce resource consumption, and allows you to create transactions that span multiple datasets.

Like other database connection components, *TADOConnection* provides support for

- Controlling connections
- Controlling server login
- Managing transactions
- Working with associated datasets
- Sending commands to the server
- Obtaining metadata

In addition to these features that are common to all database connection components, *TADOConnection* provides its own support for

- A wide range of options you can use to fine-tune the connection.
- The ability to list the command objects that use the connection.
- Additional events when performing common tasks.

Connecting to a data store using TADOConnection

One or more ADO dataset and command components can share a single connection to a data store by using *TADOConnection*. To do so, associated dataset and command components with the connection component through their *Connection* properties. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the Object Inspector. At runtime, assign the reference to the *Connection* property. For example, the following line associates a *TADODataSet* component with a *TADOConnection* component.

```
ADODataSet1.Connection := ADOConnection1;
```

The connection component represents an ADO connection object. Before you can use the connection object to establish a connection, you must identify the data store to which you want to connect. Typically, you provide information using the *ConnectionString* property. *ConnectionString* is a semicolon delimited string that lists one or more named connection parameters. These parameters identify the data store by specifying either the name of a file that contains the connection information or the name of an ADO provider and a reference identifying the data store. Use the following, predefined parameter names to supply this information:

Parameter	Description
Provider	The name of a local ADO provider to use for the connection.
Data Source	The name of the data store.
File name	The name of a file containing connection information.
Remote Provider	The name of an ADO provider that resides on a remote machine.
Remote Server	The name of the remote server when using a remote provider.

Thus, a typical value of *ConnectionString* has the form

```
Provider=MSDASQL.1;Data Source=MQIS
```

Note The connection parameters in *ConnectionString* do not need to include the *Provider* or *Remote Provider* parameter if you specify an ADO provider using the *Provider* property. Similarly, you do not need to specify the *Data Source* parameter if you use the *DefaultDatabase* property.

In addition, to the parameters listed above, *ConnectionString* can include any connection parameters peculiar to the specific ADO provider you are using. These additional connection parameters can include user ID and password if you want to hardcode the login information.

At design-time, you can use the Connection String Editor to build a connection string by selecting connection elements (like the provider and server) from lists. Click the ellipsis button for the *ConnectionString* property in the Object Inspector to launch the Connection String Editor, which is an ActiveX property editor supplied by ADO.

Once you have specified the *ConnectionString* property (and, optionally, the *Provider* property), you can use the ADO connection component to connect to or disconnect from the ADO data store, although you may first want to use other properties to fine-tune the connection. When connecting to or disconnecting from the data store, *TADOConnection* lets you respond to a few additional events beyond those common to all database connection components. These additional events are described in "Events when establishing a connection" on page 21-7 and "Events when disconnecting" on page 21-8.

Note If you do not explicitly activate the connection by setting the connection component's *Connected* property to *True*, it automatically establishes the connection when the first dataset component is opened or the first time you use an ADO command component to execute a command.

Accessing the connection object

Use the *ConnectionObject* property of *TADOConnection* to access the underlying ADO connection object. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object.

Using the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in particular. It is not recommended that you use the Connection object unless you are familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

Fine-tuning a connection

One advantage of using *TADOConnection* for establishing the connection to a data store instead of simply supplying a connection string for your ADO command and dataset components, is that it provides a greater degree of control over the conditions and attributes of the connection.

Forcing asynchronous connections

Use the *ConnectOptions* property to force the connection to be asynchronous. Asynchronous connections allow your application to continue processing without waiting for the connection to be completely opened.

By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*.

The example routines below enable and disable asynchronous connections in the specified connection component:

```
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
  end;
end;
```

```
procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
```

Controlling timeouts

You can control the amount of time that can elapse before attempted commands and connections are considered failed and are aborted using the *ConnectionTimeout* and *CommandTimeout* properties.

ConnectionTimeout specifies the amount of time, in seconds, before an attempt to connect to the data store times out. If the connection does not successfully compile prior to expiration of the time specified in *ConnectionTimeout*, the connection attempt is canceled:

```
with ADOConnection1 do begin
  ConnectionTimeout := 10 {seconds};
  Open;
end;
```

CommandTimeout specifies the amount of time, in seconds, before an attempted command times out. If a command initiated by a call to the *Execute* method does not successfully complete prior to expiration of the time specified in *CommandTimeout*, the command is canceled and ADO generates an exception:

```
with ADOConnection1 do begin
  CommandTimeout := 10 {seconds};
  Execute('DROP TABLE Employee1997', cmdText, []);
end;
```

Indicating the types of operations the connection supports

ADO connections are established using a specific mode, similar to the mode you use when opening a file. The connection mode determines the permissions available to the connection, and hence the types of operations (such as reading and writing) that can be performed using that connection.

Use the *Mode* property to indicate the connection mode. The possible values are listed in Table 21.2:

Table 21.2 ADO connection modes

Connect Mode	Meaning
cmUnknown	Permissions are not yet set for the connection or cannot be determined.
cmRead	Read-only permissions are available to the connection.
cmWrite	Write-only permissions are available to the connection.
cmReadWrite	Read/write permissions are available to the connection.
cmShareDenyRead	Prevents others from opening connections with read permissions.
cmShareDenyWrite	Prevents others from opening connection with write permissions.
cmShareExclusive	Prevents others from opening connection.
cmShareDenyNone	Prevents others from opening connection with any permissions.

The possible values for *Mode* correspond to the *ConnectModeEnum* values of the *Mode* property on the underlying ADO connection object. See the Microsoft Data Access SDK help for more information on these values.

Specifying whether the connection automatically initiates transactions

Use the *Attributes* property to control the connection component's use of retaining commits and retaining aborts. When the connection component uses retaining commits, then every time your application commits a transaction, a new transaction is automatically started. When the connection component uses retaining aborts, then every time your application rolls back a transaction, a new transaction is automatically started.

Attributes is a set that can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. When *Attributes* contains *xaCommitRetaining*, the connection uses retaining commits. When *Attributes* contains *xaAbortRetaining*, it uses retaining aborts.

Check whether either retaining commits or retaining aborts is enabled using the *in* operator. Enable retaining commits or aborts by adding the appropriate value to the attributes property; disable them by subtracting the value. The example routines below respectively enable and disable retaining commits in an ADO connection component.

```
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
  end;
end;

procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
```

Accessing the connection's commands

Like other database connection components, you can access the datasets associated with the connection using the *DataSets* and *DataSetCount* properties. However, *ADOExpress* also includes *TADOCommand* objects, which are not datasets, but which maintain a similar relationship to the connection component.

You can use the *Commands* and *CommandCount* properties of *TADOConnection* to access the associated ADO command objects in the same way you use the *DataSets* and *DataSetCount* properties to access the associated datasets. Unlike *DataSets* and *DataSetCount*, which only list active datasets, *Commands* and *CommandCount* provide references to all *TADOCommand* components associated with the connection component.

Commands is a zero-based array of references to ADO command components. *CommandCount* provides a total count of all of the commands listed in *Commands*. You can use theses properties together to iterate through all the commands that use a connection component, as illustrated in the following code*:*

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
end;
```

ADO connection events

In addition to the usual events that occur for all database connection components, *TADOConnection* generates a number of additional events that occur during normal usage.

Events when establishing a connection

In addition to the *BeforeConnect* and *AfterConnect* events that are common to all database connection components, *TADOConnection* also generates an *OnWillConnect* and *OnConnectComplete* event when establishing a connection. These events occur after the *BeforeConnect* event.

- *OnWillConnect* occurs before the ADO provider establishes a connection. It lets you make last minute changes to the connection string, provide a user name and password if you are handling your own login support, force an asynchronous connection, or even cancel the connection before it is opened.
- *OnConnectComplete* occurs after the connection is opened. Because *TADOConnection* can represent asynchronous connections, you should use *OnConnectComplete*, which occurs after the connection is opened or has failed due to an error condition, instead of the *AfterConnect* event, which occurs after the connection component instructs the ADO provider to open a connection, but not necessarily after the connection is opened.

Events when disconnecting

In addition to the *BeforeDisconnect* and *AfterDisconnect* events common to all database connection components, *TADOConnection* also generates an *OnDisconnect* event after closing a connection. *OnDisconnect* occurs after the connection is closed but before any associated datasets are closed and before the *AfterDisconnect* event.

Events when managing transactions

The ADO connection component provides a number of events for detecting when transaction-related processes have been completed. These events indicate when a transaction process initiated by a *BeginTrans*, *CommitTrans*, and *RollbackTrans* method has been successfully completed at the data store.

- The *OnBeginTransComplete* event occurs when the data store has successfully started a transaction after a call to the *BeginTrans* method.
- The *OnCommitTransComplete* event occurs after a transaction is successfully committed due to a call to *CommitTrans*.
- The *OnRollbackTransComplete* event occurs after a transaction is successfully aborted due to a call to *RollbackTrans*.

Other events

ADO connection components introduce two additional events you can use to respond to notifications from the underlying ADO connection object:

- The *OnExecuteComplete* event occurs after the connection component executes a command on the data store (for example, after calling the *Execute* method). *OnExecuteComplete* indicates whether the execution was successful.
- The *OnInfoMessage* event occurs when the underlying connection object provides detailed information after an operation is completed. The *OnInfoMessage* event handler receives the interface to an ADO Error object that contains the detailed information and a status code indicating whether the operation was successful.

Using ADO datasets

ADO dataset components encapsulate the ADO Recordset object. They inherit the common dataset capabilities described in Chapter 18, “Understanding datasets,” using ADO to provide the implementation. In order to use an ADO dataset, you must familiarize yourself with these common features.

In addition to the common dataset features, all ADO datasets add properties, events, and methods for

- Connecting to an ADO datastore.
- Accessing the underlying Recordset object.
- Filtering records based on bookmarks.
- Fetching records asynchronously.
- Performing batch updates (caching updates).
- Using files on disk to store data.

There are four ADO datasets:

- *TADOTable*, a table-type dataset that represents all of the rows and columns of a single database table. See “Using table-type datasets” on page 18-24 for information on using *TADOTable* and other table-type datasets.
- *TADOQuery*, a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See “Using query-type datasets” on page 18-41 for information on using *TADOQuery* and other query-type datasets.
- *TADOStoredProc*, a stored procedure-type dataset that executes a stored procedure defined on a database server. See “Using stored procedure-type datasets” on page 18-48 for information on using *TADOStoredProc* and other stored procedure-type datasets.
- *TADODataSet*, a general-purpose dataset that includes the capabilities of the other three types. See “Using TADODataSet” on page 21-15 for a description of features unique to *TADODataSet*.

Note When using ADO to access database information, you do not need to use a dataset such as *TADOQuery* to represent SQL commands that do not return a cursor. Instead, you can use *TADOCommand*, a simple component that is not a dataset. For details on *TADOCommand*, see “Using Command objects” on page 21-16.

Connecting an ADO dataset to a data store

ADO datasets can connect to an ADO data store either collectively or individually.

When connecting datasets collectively, set the *Connection* property of each dataset to a *TADOConnection* component. Each dataset then uses the ADO connection component’s connection.

```
ADODataSet1.Connection := ADOConnection1;
ADODataSet2.Connection := ADOConnection1;
...
```

Among the advantages of connecting datasets collectively are:

- The datasets share the connection object's attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The datasets can participate in transactions.

For more information on using *TADOConnection* see "Connecting to ADO data stores" on page 21-2.

When connecting datasets individually, set the *ConnectionString* property of each dataset. Each dataset that uses *ConnectionString* establishes its own connection to the data store, independent of any other dataset connection in the application.

The *ConnectionString* property of ADO datasets works the same way as the *ConnectionString* property of *TADOConnection*: it is a set of semicolon-delimited connection parameters such as the following:

```
ADODataSet1.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
```

At design time you can use the Connection String Editor to help you build the connection string. For more information about connection strings, see "Connecting to a data store using TADOConnection" on page 21-3.

Working with record sets

The *Recordset* property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *Recordset* to directly access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. Using the recordset object directly is not recommended unless you are familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

The *RecordsetState* property indicates the current state of the underlying recordset object. *RecordsetState* corresponds to the *State* property of the ADO recordset object. The value of *RecordsetState* is either *stOpen, stExecuting,* or *stFetching*. (*TObjectState*, the type of the *RecordsetState* property, defines other values, but only *stOpen*, *stExecuting,* and *stFetching* pertain to recordsets.) A value of *stOpen* indicates that the recordset is currently idle. A value of *stExecuting* indicates that it is executing a command. A value of *stFetching* indicates that it is fetching rows from the associated table (or tables).

Use *RecordsetState* values to perform actions dependent on the current state of the dataset. For example, a routine that updates data might check the *RecordsetState* property to see whether the dataset is active and not in the process of other activities such as connecting or fetching data.

Filtering records based on bookmarks

ADO datasets support the common dataset feature of using bookmarks to mark and return to specific records. Also like other datasets, ADO datasets let you use filters to limit the available records in the dataset. ADO datasets provide an additional feature that combines these two common dataset features: the ability to filter on a set of records identified by bookmarks.

To filter on a set of bookmarks,

1. Use the *Bookmark* method to mark the records you want to include in the filtered dataset.
2. Call the *FilterOnBookmarks* method to filter the dataset so that only the bookmarked records appear.

This process is illustrated below:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BM1, BM2: TBookmarkStr;
begin
  with ADODataSet1 do begin
    BM1 := Bookmark;
    BMList.Add(Pointer(BM1));
    MoveBy(3);
    BM2 := Bookmark;
    BMList.Add(Pointer(BM2));
    FilterOnBookmarks([BM1, BM2]);
  end;
end;
```

Note that the example above also adds the bookmarks to a list object named BMList. This is necessary so that the application can later free the bookmarks when they are no longer needed.

For details on using bookmarks, see “Marking and returning to records” on page 18-9. For details on other types of filters, see “Displaying and editing a subset of data using filters” on page 18-12.

Fetching records asynchronously

Unlike other datasets, ADO datasets can fetch their data asynchronously. This allows your application to continue performing other tasks while the dataset populates itself with data from the data store.

To control whether the dataset fetches data asynchronously, if it fetches data at all, use the *ExecuteOptions* property. *ExecuteOptions* governs how the dataset fetches its records when you call *Open* or set *Active* to *True*. If the dataset represents a query or stored procedure that does not return any records, ExecuteOptions governs how the query or stored procedure is executed when you call *ExecSQL* or *ExecProc*.

ExecuteOptions is a set that includes zero or more of the following values:

Table 21.3 Execution options for ADO datasets

Execute Option	Meaning
eoAsyncExecute	The command or data fetch operation is executed asynchronously.
eoAsyncFetch	The dataset first fetches the number of records specified by the *CacheSize* property synchronously, then fetches any remaining rows asynchronously.
eoAsyncFetchNonBlocking	Asynchronous data fetches or command execution do not block the current thread of execution.
eoExecuteNoRecords	A command or stored procedure that does not return data. If any rows are retrieved, they are discarded and not returned.

Using batch updates

One approach for caching updates is to connect the ADO dataset to a client dataset using a dataset provider. This approach is discussed in "Using a client dataset to cache updates" on page 23-15.

However, ADO dataset components provide their own support for cached updates, which they call batch updates. The following table lists the correspondences between caching updates using a client dataset and using the batch updates features:

Table 21.4 Comparison of ADO and client dataset cached updates

ADO dataset	TClientDataSet	Description
LockType	Not used: client datasets always cache updates	Specifies whether the dataset is opened in batch update mode.
CursorType	Not used: client datasets always work with an in-memory snapshot of data	Specifies how isolated the ADO dataset is from changes on the server.
RecordStatus	UpdateStatus	Indicates what update, if any, has occurred on the current row. *RecordStatus* provides more information than *UpdateStatus*.
FilterGroup	StatusFilter	Specifies which type of records are available. *FilterGroup* provides a wider variety of information.
UpdateBatch	ApplyUpdates	Applies the cached updates back to the database server. Unlike *ApplyUpdates*, *UpdateBatch* lets you limit the types of updates to be applied.
CancelBatch	CancelUpdates	Discards pending updates, reverting to the original values. Unlike *CancelUpdates*, *CancelBatch* lets you limit the types of updates to be canceled.

Using the batch updates features of ADO dataset components is a matter of:

- Opening the dataset in batch update mode
- Inspecting the update status of individual rows
- Filtering multiple rows based on update status
- Applying the batch updates to base tables
- Canceling batch updates

Opening the dataset in batch update mode

To open an ADO dataset in batch update mode, it must meet these criteria:

1. The component's *CursorType* property must be *ctKeySet* (the default property value) or *ctStatic*.
2. The *LockType* property must be *ltBatchOptimistic*.
3. The command must be a SELECT query.

Before activating the dataset component, set the *CursorType* and *LockType* properties as indicated above. Assign a SELECT statement to the component's *CommandText* property (for *TADODataSet*) or the *SQL* property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that returns a result set. These properties can be set at design-time through the Object Inspector or programmatically at runtime. The example below shows the preparation of a *TADODataSet* component for batch update mode.

```
with ADODataSet1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

After a dataset has been opened in batch update mode, all changes to the data are cached rather than applied directly to the base tables.

Inspecting the update status of individual rows

Determine the update status of a given row by making it current and then inspecting the *RecordStatus* property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unchanged record';
  rsModified:   StatusBar1.Panels[0].Text := 'Changed record';
  rsDeleted:    StatusBar1.Panels[0].Text := 'Deleted record';
  rsNew:        StatusBar1.Panels[0].Text := 'New record';
end;
```

Filtering multiple rows based on update status

Filter a recordset to show only those rows that belong to a group of rows with the same update status using the *FilterGroup* property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

Note For the *FilterGroup* property to have an effect, the ADO dataset component's *Filtered* property must be set to *True*.

Applying the batch updates to base tables

Apply pending data changes that have not yet been applied or canceled by calling the *UpdateBatch* method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache.

Used alone with no parameter, *UpdateBatch* applies all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *arAll* is passed, only a subset of the pending changes are applied. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be applied. The example below applies only the currently active row to be applied:

```
ADODataSet1.UpdateBatch(arCurrent);
```

Canceling batch updates

Cancel pending data changes that have not yet been canceled or applied by calling the *CancelBatch* method. When you cancel pending batch updates, field values on rows that have been changed revert to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes.

Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *CancelBatch*. If any value except *arAll* is passed, only a subset of the pending changes are canceled. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
ADODataSet1.CancelBatch;
```

Loading data from and saving data to files

The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. The data is saved in one of two proprietary formats: ADTG or XML. These two file formats are the only formats supported by ADO. However, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the version you are using to determine what save file formats are supported.

Save the data to a file using the *SaveToFile* method. *SaveToFile* takes two parameters, the name of the file to which data is saved, and, optionally, the format (ADTG or XML) in which to save the data. Indicate the format for the saved file by setting the *Format* parameter to *pfADTG* or *pfXML*. If the file specified by the *FileName* parameter already exists, *SaveToFile* raises an *EOleException*.

Retrieve the data from file using the *LoadFromFile* method. *LoadFromFile* takes a single parameter, the name of the file to load. If the specified file does not exist, *LoadFromFile* raises an *EOleException* exception. On calling the *LoadFromFile* method, the dataset component is automatically activated.

In the example below, the first procedure saves the dataset retrieved by the *TADODataSet* component *ADODataSet1* to a file. The target file is an ADTG file named SaveFile, saved to a local drive. The second procedure loads this saved file into the *TADODataSet* component *ADODataSet2*.

```
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
  begin
    DeleteFile('c:\SaveFile');
    StatusBar1.Panels[0].Text := 'Save file deleted!';
  end;
  ADODataSet1.SaveToFile('c:\SaveFile', pfADTG);
end;

procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataSet2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Save file does not exist!';
end;
```

The datasets that save and load the data need not be on the same form as above, in the same application, or even on the same computer. This allows for the briefcase-style transfer of data from one computer to another.

Using TADODataSet

TADODataSet is a general-purpose dataset for working with data from an ADO data store. Unlike the other ADO dataset components, *TADODataSet* is not a table-type, query-type, or stored procedure-type dataset. Instead, it can function as any of these types:

- Like a table-type dataset, *TADODataSet* lets you represent all of the rows and columns of a single database table. To use it in this way, set the *CommandType* property to *cmdTable* and the *CommandText* property to the name of the table. *TADODataSet* supports table-type tasks such as

  - Assigning indexes to sort records or form the basis of record-based searches. In addition to the standard index properties and methods described in “Sorting records with indexes” on page 18-25, *TADODataSet* lets you sort using temporary indexes by setting the *Sort* property. Indexed-based searches performed using the *Seek* method use the current index.

  - Emptying the dataset. The *DeleteRecords* method provides greater control than related methods in other table-type datasets, because it lets you specify what records to delete.

  The table-type tasks supported by *TADODataSet* are available even when you are not using a *CommandType* of *cmdTable*.

- Like a query-type dataset, *TADODataSet* lets you specify a single SQL command that is executed when you open the dataset. To use it in this way, set the *CommandType* property to *cmdText* and the *CommandText* property to the SQL command you want to execute. At design time, you can double-click on the *CommandText* property in the Object Inspector to use the Command Text editor for help in constructing the SQL command. *TADODataSet* supports query-type tasks such as
    - Using parameters in the query text. See "Using parameters in queries" on page 18-43 for details on query parameters.
    - Setting up master/detail relationships using parameters. See "Establishing master/detail relationships using parameters" on page 18-46 for details on how to do this.
    - Preparing the query in advance to improve performance by setting the *Prepared* property to *True.*
- Like a stored procedure-type dataset, *TADODataSet* lets you specify a stored procedure that is executed when you open the dataset. To use it in this way, set the *CommandType* property to *cmdStoredProc* and the *CommandText* property to the name of the stored procedure. *TADODataSet* supports stored procedure-type tasks such as
    - Working with stored procedure parameters. See "Working with stored procedure parameters" on page 18-50 for details on stored procedure parameters.
    - Fetching multiple result sets. See "Fetching multiple result sets" on page 18-53 for details on how to do this.
    - Preparing the stored procedure in advance to improve performance by setting the *Prepared* property to *True*.

In addition, *TADODataSet* lets you work with data stored in files by setting the *CommandType* property to *cmdFile* and the *CommandText* property to the file name.

Before you set the *CommandText* and *CommandType* properties, you should link the *TADODataSet* to a data store by setting the *Connection* or *ConnectionString* property. This process is described in "Connecting an ADO dataset to a data store" on page 21-9. As an alternative, you can use an RDS DataSpace object to connect the *TADODataSet* to an ADO-based application server. To use an RDS DataSpace object, set the *RDSConnection* property to a *TRDSConnection* object.

Using Command objects

In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language.

Although you can always execute commands using *TADOQuery*, you may not want the overhead of using a dataset component, especially if the command does not return a result set. As an alternative, you can use the *TADOCommand* component, which is a lighter-weight object designed to execute commands, one command at a time. *TADOCommand* is intended primarily for executing those commands that do not return result sets, such as Data Definition Language (DDL) SQL statements. Through an overloaded version of its *Execute* method, however, it is capable of returning a result set that can be assigned to the *RecordSet* property of an ADO dataset component.

In general, working with *TADOCommand* is very similar to working with *TADODataSet*, except that you can't use the standard dataset methods to fetch data, navigate records, edit data, and so on. *TADOCommand* objects connect to a data store in the same way as ADO datasets. See "Connecting an ADO dataset to a data store" on page 21-9 for details.

The following topics provide details on how to specify and execute commands using *TADOCommand*.

Specifying the command

Specify commands for a *TADOCommand* component using the *CommandText* property. Like *TADODataSet*, *TADOCommand* lets you specify the command in different ways, depending on the *CommandType* property. Possible values for *CommandType* include: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the Object Inspector. At runtime, assign a value of type *TCommandType* to the *CommandType* property.

```
with ADOCommand1 do begin
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
...
end;
```

If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*.

CommandText can contain the text of an SQL query that includes parameters or the name of a stored procedure that uses parameters. You must then supply parameter values, which are bound to the parameters before executing the command. See "Handling command parameters" on page 21-19 for details.

Using the Execute method

Before *TADOCommand* can execute its command, it must have a valid connection to a data store. This is established just as with an ADO dataset. See "Connecting an ADO dataset to a data store" on page 21-9 for details.

To execute the command, call the *Execute* method. *Execute* is an overloaded method that lets you choose the most appropriate way to execute the command.

For commands that do not require any parameters and for which you do not need to know how many records were affected, call *Execute* without any parameters:

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

Other versions of *Execute* let you provide parameter values using a Variant array, and to obtain the number of records affected by the command.

For information on executing commands that return a result set, see "Retrieving result sets with commands" on page 21-18.

Canceling commands

If you are executing the command asynchronously, then after calling *Execute* you can abort the execution by calling the *Cancel* method:

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;

procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

The *Cancel* method only has an effect if there is a command pending and it was executed asynchronously (*eoAsynchExecute* is in the *ExecuteOptions* parameter of the *Execute* method). A command is said to be pending if the *Execute* method has been called but the command has not yet been completed or timed out.

A command times out if it is not completed or canceled before the number of seconds specified in the *CommandTimeout* property expire. By default, commands time out after 30 seconds.

Retrieving result sets with commands

Unlike *TADOQuery* components, which use different methods to execute depending on whether they return a result set, *TADOCommand* always uses the *Execute* command to execute the command, regardless of whether it returns a result set. When the command returns a result set, *Execute* returns an interface to the ADO _RecordSet interface.

The most convenient way to work with this interface is to assign it to the *RecordSet* property of an ADO dataset.

For example, the following code uses *TADOCommand* (*ADOCommand1*) to execute a SELECT query, which returns a result set. This result set is then assigned to the *RecordSet* property of a *TADODataSet* component (*ADODataSet1*).

```
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataSet1.Recordset := Execute;
end;
```

As soon as the result set is assigned to the ADO dataset's *Recordset* property, the dataset is automatically activated and the data is available.

Handling command parameters

There are two ways in which a *TADOCommand* object may use parameters:

- The *CommandText* property can specify a query that includes parameters. Working with parameterized queries in *TADOCommand* works like using a parameterized query in an ADO dataset. See "Using parameters in queries" on page 18-43 for details on parameterized queries.
- The *CommandText* property can specify a stored procedure that uses parameters. Stored procedure parameters work much the same using *TADOCommand* as with an ADO dataset. See "Working with stored procedure parameters" on page 18-50 for details on stored procedure parameters.

There are two ways to supply parameter values when working with *TADOCommand*: you can supply them when you call the *Execute* method, or you can specify them ahead of time using the *Parameters* property.

The *Execute* method is overloaded to include versions that take a set of parameter values as a Variant array. This is useful when you want to supply parameter values quickly without the overhead of setting up the *Parameters* property:

```
ADOCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
```

When working with stored procedures that return output parameters, you must use the *Parameters* property instead. Even if you do not need to read output parameters, you may prefer to use the *Parameters* property, which lets you supply parameters at design time and lets you work with *TADOCommand* properties in the same way you work with the parameters on datasets.

When you set the *CommandText* property, the *Parameters* property is automatically updated to reflect the parameters in the query or those used by the stored procedure. At design-time, you can use the Parameter Editor to access parameters, by clicking the ellipsis button for the *Parameters* property in the Object Inspector. At runtime, use properties and methods of *TParameter* to set (or get) the values of each parameter.

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

Chapter

22

Using unidirectional datasets

dbExpress is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy a database application that uses *dbExpress*, you need only include a dll (the server-specific driver) with the application files you build.

dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by *TDataSet* are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the *First* and *Next* methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The *CanModify* property is always *False*, so attempts to put the dataset into edit mode always fail. You can, however, use unidirectional datasets to update data using an SQL UPDATE command or provide conventional editing support by using a dbExpress-enabled client dataset or connecting the dataset to a client dataset (see “Connecting to another dataset” on page 14-10).
- There is no support for filters, because filters work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.

- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are the fastest data access mechanism, and very simple to use and deploy.

Types of unidirectional datasets

The *dbExpress* page of the component palette contains four types of unidirectional dataset: *TSQLDataSet*, *TSQLQuery*, *TSQLTable*, and *TSQLStoredProc.*

TSQLDataSet is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

TSQLQuery is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any. See “Using query-type datasets” on page 18-41 for information on using query-type datasets.

TSQLTable is a table-type dataset that represents all of the rows and columns of a single database table. See “Using table-type datasets” on page 18-24 for information on using table-type datasets.

TSQLStoredProc is a stored procedure-type dataset that executes a stored procedure defined on a database server. See “Using stored procedure-type datasets” on page 18-48 for information on using stored procedure-type datasets.

Note The *dbExpress* page also includes *TSQLClientDataSet*, which is not a unidirectional dataset. Rather, it is a client dataset that uses a unidirectional dataset internally to access its data

Connecting to the database server

The first step when working with a unidirectional dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the Object Inspector can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the Object Inspector can list what stored procedures are available on the server.

The connection to a database server is represented by a separate *TSQLConnection* component. You work with *TSQLConnection* like any other database connection component. For information about database connection components, see Chapter 17, “Connecting to databases”.

To use *TSQLConnection* to connect a unidirectional dataset to a database server, set the *SQLConnection* property. At design time, you can choose the SQL connection component from a drop-down list in the Object Inspector. If you make this assignment at runtime, be sure that the connection is active:

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers. However, you may want to use a separate connection for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the *MaxStmtsPerConn* property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the *AutoClone* property to *False*.

Before you assign the *SQLConnection* property, you will need to set up the *TSQLConnection* component so that it identifies the database server and any required connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on).

Setting up TSQLConnection

In order to describe a database connection in sufficient detail for *TSQLConnection* to open a connection, you must identify both the driver to use and a set of connection parameters the are passed to that driver.

Identifying the driver

The driver is identified by the *DriverName* property, which is the name of an installed *dbExpress* driver, such as INTERBASE, ORACLE, MYSQL, or DB2. The driver name is associated with two files

- The *dbExpress* driver. This can be either a dynamic-link library with a name like dbexpint.dll, dbexpora.dll, dbexpmys.dll, or dbexpdb2.dll, or a compiled unit that you can statically link into your application (dbexptint.dcu, dbexpora.dcu, dbexpmys.dcu, or dbexpdb2.dcu).
- The dynamic-link library provided by the database vendor for client-side support.

The relationship between these two files and the database name is stored in a file called dbxdrivers.ini, which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in dbxdrivers.ini when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the *LibraryName* and *VendorLib* properties to the names of the associated dlls. Once *LibraryName* and *VendorLib* have been set, your application does not need to rely on dbxdrivers.ini. (That is, you do not need to deploy dbxdrivers.ini with your application unless you set the *DriverName* property at runtime.)

Specifying connection parameters

The *Params* property is a string list that lists name/value pairs. Each pair has the form *Name=Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with ORACLE it is the entry in TNSNames.ora, while with DB2, it is the client-side node name.

Other typical parameters include the *User_Name* (the name to use when logging in), *Password* (the password for *User_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the Object Inspector to edit the parameters using the String List editor. At runtime, use the *Params.Values* property to assign values to individual parameters.

Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. You can name *dbExpress* database and parameter combinations, which are then saved in a file called dbxconnections.ini. The name of each combination is called a connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the *ConnectionName* property to a valid connection name. Setting *ConnectionName* automatically sets the *DriverName* and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the dbxconnections.ini file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of dbxconnections.ini. Then, when you deploy your application, it loads these values from a separate version of dbxconnections.ini that uses the “real” database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the *LoadParamsOnConnect* property to *True*. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini when the connection is opened.

- Call the *LoadParamsFromIniFile* method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the dbxconnections.ini file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in dbxconnections.ini:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to dbxconnections.ini.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to dbxconnections.ini.
- Click the Delete Connection button to delete the currently selected named connection from dbxconnections.ini.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

Specifying what data to display

There are a number of ways to specify what data a unidirectional dataset represents. Which method you choose depends on the type of unidirectional dataset you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a *TSQLDataSet* component, use the *CommandType* property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.

- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

Note You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see “Fetching metadata into a unidirectional dataset” on page 22-12.

Representing the results of a query

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

When using *TSQLDataSet*, set the *CommandType* property to *ctQuery* and assign the text of the query statement to the *CommandText* property. When using *TSQLQuery*, assign the query to the *SQL* property instead. These properties work the same way for all general-purpose or query-type datasets. “Specifying the query” on page 18-42 discusses them in greater detail.

When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. Using parameters in queries and supplying values for those parameters is discussed in “Using parameters in queries” on page 18-43.

SQL defines queries such as UPDATE queries that perform actions on the server but do not return records. Such queries are discussed in “Executing commands that do not return records” on page 22-9.

Representing the records in a table

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

Note If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This can result in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields on the server.

Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the *CommandType* property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- *CommandText* specifies the name of the database table that the *TSQLDataSet* object should represent.
- *SortFieldNames* lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

TSQLDataSet generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the *TableName* property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the *IndexName* property to the name of an index defined on the server that imposes the order you want.
- Set the *IndexFieldNames* property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

Representing the results of a stored procedure

Stored procedures are sets of SQL statements that are named and stored on an SQL server. How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

When using *TSQLDataSet*, to specify a stored procedure:

- Set the *CommandType* property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the *CommandText* property:

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

When using *TSQLStoredProc* , you need only specify the name of the stored procedure as the value of the *StoredProcName* property.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure. See “Working with stored procedure parameters” on page 18-50 for information about working with stored procedure parameters.

Fetching the data

Once you have specified the source of the data, you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a unidirectional dataset:

- Set the *Active* property to *True*, either at design time in the Object Inspector, or in code at runtime:

  ```
  CustQuery.Active := True;
  ```

- Call the *Open* method at runtime,

  ```
  CustQuery.Open;
  ```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

Preparing the dataset

Before a query or stored procedure can execute on the server, it must first be “prepared”. Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to *True* or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its *Prepared* property to *True*.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

Fetching multiple datasets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the *NextRecordSet* method:

```
var
  DataSet2: TSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLDataSet1.NextRecordSet(nRows);
  ...
```

NextRecordSet returns a newly created *TSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns **nil**.

Executing commands that do not return records

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

Note If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See “Sending commands to the server” on page 17-10 for details.

Specifying the command to execute

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using *TSQLDataSet*, use the *CommandType* and *CommandText* properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.

- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using *TSQLQuery*, use the *SQL* property to specify the SQL statement to pass to the server.

When using *TSQLStoredProc*, use the *StoredProcName* property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters or stored procedure parameters the same way as with queries and stored procedures that return records. See "Using parameters in queries" on page 18-43 and "Working with stored procedure parameters" on page 18-50 for details.

Executing the command

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

- The *ExecSQL* method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
```

- The *ExecProc* method if the dataset is an instance of *TSQLStoredProc*.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
```

Tip If you are executing the query or stored procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

Creating and modifying server metadata

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset and let it handle all the generation of all SQL commands concerned with editing (see "Connecting a client dataset to another dataset in the same application" on page 14-11). In fact, this is the recommended approach because data-aware controls are designed to perform edits through a dataset such as *TClientDataSet*.

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET_EMP_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP_NO and :PROJ_ID) with a parameter of the query that creates the stored procedure.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
       END';
  ExecSQL;
end;
```

Setting up master/detail linked cursors

There are two ways to use linked cursors to set up a master/detail relationship with a unidirectional dataset as the detail set. Which method you use depends on the type of unidirectional dataset you are using. Once you have set up such a relationship, the unidirectional dataset (the "many" in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the "one" in the one-to-many relationship).

TSQLDataSet and *TSQLQuery* require you to use a parameterized query to establish a master/detail relationship. This is the technique for creating such relationships on all query-type datasets. For details on creating master/detail relationships with query-type datasets, see "Establishing master/detail relationships using parameters" on page 18-46.

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the *MasterSource* and *MasterFields* properties, just as you would with any other table-type dataset. For details on creating master/detail relationships with table-type datasets, see "Establishing master/detail relationships using parameters" on page 18-46.

Accessing schema information

There are two ways to obtain information about what is available on the server. This information, called schema information or metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

The simplest way to obtain this metadata is to use the methods of *TSQLConnection*. These methods fill an existing string list or list object with the names of tables, stored procedures, fields, or indexes, or with parameter descriptors. This technique is the same as the way you fill lists with metadata for any other database connection component. These methods are described in "Obtaining metadata" on page 17-12.

If you require more detailed schema information, you can populate a unidirectional dataset with metadata. Instead of a simple list, the unidirectional dataset is filled with schema information, where each record represents a single table, stored procedure, index, field, or parameter.

Fetching metadata into a unidirectional dataset

To populate a unidirectional datasets with metadata from the database server, you must first indicate what data you want to see, using the *SetSchemaInfo* method. *SetSchemaInfo* takes three parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see “The structure of metadata datasets” on page 22-13.

- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil.
- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as ‘Cust%’, which uses the wildcards ‘%’ (to match a string of arbitrary characters of any length) and ‘_’ (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or __). If you do not want to use a pattern, this parameter can be nil.

Note If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection’s *TableScope* property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, Instead, you want to obtain a list of input parameters for a stored procedure named ‘MyProc’. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters (‘inName’, ‘outValue’ and so on). You could call *SetSchemaInfo* as follows:

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the *CommandText* property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

The structure of metadata datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

Table 22.1 Columns in tables of metadata listing tables

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the table.
TABLE_NAME	ftString	The name of the table. This field determines the sort order of the dataset.
TABLE_TYPE	ftInteger	Identifies the type of table. It is a sum of one or more of the following values: 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table.

Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

Table 22.2 Columns in tables of metadata listing stored procedures

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure. This field determines the sort order of the dataset.
PROC_TYPE	ftInteger	Identifies the type of stored procedure. It is a sum of one or more of the following values: 1: Procedure 2: Function 4: Package 8: System procedure
IN_PARAMS	ftSmallint	The number of input parameters
OUT_PARAMS	ftSmallint	The number of output parameters.

Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

Table 22.3 Columns in tables of metadata listing fields

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table whose fields you listing. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the field.
TABLE_NAME	ftString	The name of the table that contains the fields.
COLUMN_NAME	ftString	The name of the field. This value determines the sort order of the dataset.
COLUMN_POSITION	ftSmallint	The position of the column in its table.
COLUMN_TYPE	ftInteger	Identifies the type of value in the field. It is a sum of one or more of the following: 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	The datatype of the column. This is one of the logical field type constants defined in sqllinks.pas.
COLUMN_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in COLUMN_DATATYPE and COLUMN_SUBTYPE, but in a form used in some DDL statements.
COLUMN_SUBTYPE	ftSmallint	A subtype for the column's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
COLUMN_PRECISION	ftInteger	The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on).
COLUMN_SCALE	ftSmallint	The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields.
COLUMN_LENGTH	ftInteger	The number of bytes required to store field values.
COLUMN_NULLABLE	ftSmallint	A Boolean that indicates whether the field can be left blank (0 means the field requires a value).

Information about indexes

When you request information about the indexes on a table (stIndexes), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

Table 22.4 Columns in tables of metadata listing indexes

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the index. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the index.
TABLE_NAME	ftString	The name of the table for which the index is defined.
INDEX_NAME	ftString	The name of the index. This field determines the sort order of the dataset.
PKEY_NAME	ftString	Indicates the name of the primary key.
COLUMN_NAME	ftString	The name of the field (column) in the index.
COLUMN_POSITION	ftSmallint	The position of this field in the index.
INDEX_TYPE	ftSmallint	Identifies the type of index. It is a sum of one or more of the following values: 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	Indicates that the index is ascending (a) or descending (d).
FILTER	ftString	Describes a filter condition that limits the indexed records.

Information about stored procedure parameters

When you request information about the parameters of a stored procedure (*stProcedureParams*), the resulting dataset includes a record for each parameter. It has the following columns:

Table 22.5 Columns in tables of metadata listing parameters

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure that contains the parameter.
PARAM_NAME	ftString	The name of the parameter. This field determines the sort order of the dataset.
PARAM_TYPE	ftSmallint	Identifies the type of parameter. This is the same as a *TParam* object's *ParamType* property.

Table 22.5 Columns in tables of metadata listing parameters (continued)

Column name	Field type	Contents
PARAM_DATATYPE	ftSmallint	The datatype of the parameter. This is one of the logical field type constants defined in sqllinks.pas.
PARAM_SUBTYPE	ftSmallint	A subtype for the parameter's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
PARAM_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in PARAM_DATATYPE and PARAM_SUBTYPE, but in a form used in some DDL statements.
PARAM_PRECISION	ftInteger	The maximum number of digits in floating-point values or bytes (for strings and Bytes fields).
PARAM_SCALE	ftSmallint	The number of digits to the right of the decimal on floating-point values.
PARAM_LENGTH	ftInteger	The number of bytes required to store parameter values.
PARAM_NULLABLE	ftSmallint	A Boolean that indicates whether the parameter can be left blank (0 means the parameter requires a value).

Debugging dbExpress applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

Using TSQLMonitor to monitor SQL commands

TSQLConnection uses a companion component, *TSQLMonitor*, to intercept these messages and save them in a string list. *TSQLMonitor* works much like the SQL monitor utility that you can use with the BDE, except that it monitors only those commands involving a single *TSQLConnection* component rather than all commands managed by *dbExpress*.

To use *TSQLMonitor*,

1. Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.
2. Set its *SQLConnection* property to the *TSQLConnection* component.
3. Set the SQL monitor's *Active* property to *True*.

As SQL commands are sent to the server, the SQL monitor's *TraceList* property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the *FileName* property and then setting the *AutoSave* property to *True*. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time is logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the *OnLogTrace* event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of *TraceList* every 10th message, clearing the log after saving it so that the list never gets too long:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
  end;
end;
```

Note If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

Using a callback to monitor SQL commands

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's *SetTraceCallbackEvent* method. *SetTraceCallbackEvent* takes two parameters: a callback of type *TSQLCallbackEvent*, and a user-defined value that is passed to the callback function.

The callback function takes two parameters: *CallType* and *CBInfo*:

- *CallType* is reserved for future use.
- *CBInfo* is a pointer to a record that includes the category (the same as *CallType*), the text of the SQL command, and the user-defined value that is passed to the *SetTraceCallbackEvent* method.

The callback returns a value of type *CBRType*, typically *cbrUSEDEF*.

The *dbExpress* driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

Warning Do not call *SetTraceCallbackEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

Chapter

23

Using client datasets

Client datasets are specialized datasets that hold all their data in memory. The support for manipulating the data they store in memory is provided by midaslib.dcu or midas.dll. The format client datasets use for storing data is self-contained and easily transported, which allows client datasets to

- Read from and write to dedicated files on disk, acting as a file-based dataset. Properties and methods supporting this mechanism are described in "Using a client dataset with file-based data" on page 23-31.
- Cache updates for data from a database server. Client dataset features that support cached updates are described in "Using a client dataset to cache updates" on page 23-15.
- Represent the data in the client portion of a multi-tiered application. To function in this way, the client dataset must work with an external provider, as described in "Using a client dataset with a provider" on page 23-23. For information about multi-tiered database applications, see Chapter 25, "Creating multi-tiered applications."
- Represent the data from a source other than a dataset. Because a client dataset can use the data from an external provider, specialized providers can adapt a variety of information sources to work with client datasets. For example, you can use an XML provider to enable a client dataset to represent the information in an XML document.

Whether you use client datasets for file-based data, caching updates, data from an external provider (such as working with an XML document or in a multi-tiered application), or a combination of these approaches such as a "briefcase model" application, you can take advantage of broad range of features client datasets support for working with data.

Working with data using a client dataset

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Chapter 15, “Using data controls”for information on how to display database information in data-aware controls.

Client datasets implement all the properties an methods inherited from *TDataSet*. For a complete introduction to this generic dataset behavior, see Chapter 18, “Understanding datasets.”

In addition, client datasets implement many of the features common to table-type datasets such as

- Sorting records with indexes.
- Using Indexes to search for records.
- Limiting records with ranges.
- Creating master/detail relationships.
- Controlling read/write access
- Creating the underlying dataset
- Emptying the dataset
- Synchronizing client datasets

For details on these features, see “Using table-type datasets” on page 18-24.

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for some database functions can involve additional capabilities or considerations. This chapter describes some of these common functions and the differences introduced by client datasets.

Navigating data in client datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset’s records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see “Navigating datasets” on page 18-5.

Unlike most datasets, client datasets can also position the cursor at a specific record in the dataset by using the *RecNo* property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a particular record number to make that record the current one.

Limiting what records appear

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see “Displaying and editing a subset of data using filters” on page 18-12. For more information about ranges, see “Limiting records with ranges” on page 18-30.

With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because of this, the SQL dialect of the server limits what operations are used in filter strings. Client datasets implement their own filter support, which includes more operations than that of other datasets. For example, when using a client dataset, filter expressions can include string operators that return substrings, operators that parse date/time values, and much more. Client datasets also allow filters on BLOB fields or complex field types such as ADT fields and array fields.

The various operators and functions that client datasets can use in filters, along with a comparison to other datasets that support filters, is given below:

Table 23.1 Filter support in client datasets

Operator or function	Example	Supported by other datasets	Comment
Comparisons			
=	State = 'CA'	Yes	
<>	State <> 'CA'	Yes	
>=	DateEntered >= '1/1/1998'	Yes	
<=	Total <= 100,000	Yes	
>	Percentile > 50	Yes	
<	Field1 < Field2	Yes	
BLANK	State <> 'CA' or State = BLANK	Yes	Blank records do not appear unless explicitly included in the filter.
IS NULL	Field1 IS NULL	No	
IS NOT NULL	Field1 IS NOT NULL	No	
Logical operators			
and	State = 'CA' and Country = 'US'	Yes	
or	State = 'CA' or State = 'MA'	Yes	
not	not (State = 'CA')	Yes	
Arithmetic operators			
+	Total + 5 > 100	Depends on driver	Applies to numbers, strings, or date (time) + number.
-	Field1 - 7 <> 10	Depends on driver	Applies to numbers, dates, or date (time) - number.
*	Discount * 100 > 20	Depends on driver	Applies to numbers only.
/	Discount > Total / 5	Depends on driver	Applies to numbers only.

Table 23.1 Filter support in client datasets (continued)

Operator or function	Example	Supported by other datasets	Comment
String functions			
Upper	Upper(Field1) = 'ALWAYS'	No	
Lower	Lower(Field1 + Field2) = 'josp'	No	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	No	Value goes from position of second argument to end or number of chars in third argument. First char has position 1.

Trim	Trim(Field1 + Field2)
	Trim(Field1, '-')
		No	Removes third argument from front and back. If no third argument, trims spaces.

TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '$') <> ''	No	See Trim.
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> ''	No	See Trim.
DateTime functions			
Year	Year(DateField) = 2000	No	
Month	Month(DateField) <> 12	No	
Day	Day(DateField) = 1	No	
Hour	Hour(DateField) < 16	No	
Minute	Minute(DateField) = 0	No	
Second	Second(DateField) = 30	No	
GetDate	GetDate - DateField > 7	No	Represents current date and time.
Date	DateField = Date(GetDate)	No	Returns the date portion of a datetime value.
Time	TimeField > Time(GetDate)	No	Returns the time portion of a datetime value.
Miscellaneous			
Like	Memo LIKE '%filters%'	No	Works like SQL-92 without the ESC clause. When applied to BLOB fields, FilterOptions determines whether case is considered.
In	Day(DateField) in (1,7)	No	Works like SQL-92. Second argument is a list of values all with the same type.
*	State = 'M*'	Yes	Wildcard for partial comparisons.

When applying ranges or filters, the client dataset still stores all of its records in memory. The range or filter merely determines which records are available to controls that navigate or display data from the client dataset.

Note When fetching data from a provider, you can also limit the data that the client dataset stores by supplying parameters to the provider. For details, see "Limiting records with parameters" on page 23-28.

Editing data

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's *Data* property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the *Delta* property. Using a change log serves two purposes:

- The change log is required for applying updates to a database server or external provider component.
- The change log provides sophisticated support for undoing changes.

The *LogChanges* property lets you disable logging. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in file-based applications if you do not want the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

Note Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

Undoing changes

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call *UndoLastChange*. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (*True*), or to leave the cursor on the current record (*False*). If there are several changes to a record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure. If the removal occurs, *UndoLastChange* returns *True*. Use the *ChangeCount* property to check whether there are more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.

- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call *RevertRecord*. *RevertRecord* removes any changes to the current record from the change log.
- To restore a deleted record, first set the *StatusFilter* property to [*usDeleted*], which makes the deleted records "visible." Next, navigate to the record you want to restore and call *RevertRecord*. Finally, restore the *StatusFilter* property to [*usModified*, *usInserted*, *usUnmodified*] so that the edited version of the dataset (now containing the restored record) is again visible.
- At any point during edits, you can save the current state of the change log using the *SavePoint* property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling *CancelUpdates*. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

Saving changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client datasets stores its data in a file or represents data obtained through a provider. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. "Merging changes into data" on page 23-33 describes this process.

You can't use *MergeChangeLog* if you are using the client dataset to cache updates or to represent the data from an external provider component. The information in the change log is required fro resolving updated records with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which attempts to write the modifications to the database server or source dataset, and updates the *Data* property only when the modifications have been successfully committed. See "Applying updates" on page 23-19 for more information about this process.

Constraining data values

Client datasets can enforce constraints on the edits a user makes to data. These constraints are applied when the user tries to post changes to the change log. You can always supply custom constraints. These let you provide your own, application-defined limits on what values users post to a client dataset.

In addition, when client datasets represent server data that is accessed using the BDE, they also enforce data constraints imported from the database server. If the client dataset works with an external provider component, the provider can control whether those constraints are sent to the client dataset, and the client dataset can control whether it uses them. For details on how the provider controls whether constraints are included in data packets, see “Handling server constraints” on page 24-12. For details on how and why client dataset can turn off enforcement of server constraints, see “Handling constraints from the server” on page 23-28.

Specifying custom constraints

You can use the properties of the client dataset’s field components to impose your own constraints on what data users can enter. Each field component has two properties that you can use to specify constraints:

- The *DefaultExpression* property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset’s version takes precedence because it is assigned before the update is applied back to the database server or source dataset.
- The *CustomConstraint* property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints defined this way are applied in addition to any constraints imported from the server. For more information about working with custom constraints on field components, see “Creating a custom constraint” on page 19-21.

In addition, you can create record-level constraints using the client dataset’s *Constraints* property. *Constraints* is a collection of *TCheckConstraint* objects, where each object represents a separate condition. Use the *CustomConstraint* property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

Sorting and indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application set up relationships with other datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset represents server data or uses an external provider, it inherits a default index and sort order based on the data it receives. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets.

Note You may also want to review the material on indexes in table-type datasets, which also applies to client datasets. This material is in "Sorting records with indexes" on page 18-25 and "Limiting records with ranges" on page 18-30.

Adding a new index

There are three ways to add indexes to a client dataset:

- To create a temporary index at runtime that sorts the records in the client dataset, you can use the *IndexFieldNames* property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index.

  This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping. These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.

- To create an index at runtime that can be used for grouping, call *AddIndex*. *AddIndex* lets you specify the properties of the index, including

  - The name of the index. This can be used for switching indexes at runtime.
  - The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields.
  - How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can set options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order.
  - The default level of grouping support for the index.

  Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can't call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

- The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call *CreateDataSet*. See "Creating and deleting tables" on page 18-37 for more information about creating client datasets.

  As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others. Indexes created this way always persist and are saved when you save the client dataset to a file.

Tip You can index and sort on internally calculated fields with client datasets.

Deleting and switching indexes

To remove an index you created for a client dataset, call *DeleteIndex* and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index when more than one index is available, use the *IndexName* property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the Object Inspector.

Using indexes to group data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to display a field value only if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

To determine where the current record falls within any group, use the *GetGroupState* method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

Representing calculated values

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record. For more information about using calculated fields, see "Defining a calculated field" on page 19-7.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields. For more information on internally calculated fields, see "Using internally calculated fields in client datasets" below.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates. For more information on maintained aggregates, see "Using maintained aggregates" on page 23-11.

Using internally calculated fields in client datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an *OnCalcFields* event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnCalcFields* event handler. However, you can optimize your event handler by checking the *State* property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. Depending on whether you use persistent fields or field definitions, you do this in one of the following ways:

- If you use persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor.
- If you use field definitions, set the *InternalCalcField* property of the relevant field definition to *True*.

Note Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the BDE or remote database server.

Using maintained aggregates

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate."

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

Specifying aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the *Aggregates* property. *Aggregates* is a collection of aggregate specifications (*TAggregate*). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the *Add* method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

Note When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields. For details on creating aggregated persistent fields, see "Defining an aggregate field" on page 19-10.

For each aggregate, the *Expression* property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in Table 23.2

Table 23.2 Summary operators for maintained aggregates

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

```
Sum(Qty * Price)             {legal -- summary of an expression on fields }
Max(Field1) - Max(Field2)    {legal -- expression on summaries }
Avg(DiscountRate) * 100      {legal -- expression of summary and constant }
Min(Sum(Field1))             {illegal -- nested summaries }
Count(Field1) - Field2       {illegal -- expression of summary and field }
```

Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in the client dataset. However, you can specify that you want to summarize over the records in a group instead. This lets you provide intermediate summaries such as subtotals for groups of records that share a common field value.

Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping. See "Using indexes to group data" on page 23-9 for information on grouping support.

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

Obtaining aggregate values

To get the value of a maintained aggregate, call the *Value* method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the *GetGroupState* method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification. The *AggFields* property contains the new aggregated field component, and the *FindField* method returns it.

Copying data from another dataset

To copy the data from another dataset at design time, right click the client dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one whose data and structure you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Assigning data directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a client dataset that represents server data or that uses an external provider component, data packets are automatically assigned to *Data*.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

Note When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

If you are copying from a dataset other than a client dataset, you can create a dataset provider component, link it to the source dataset, and then copy its data:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

Note When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

If you want to merge changes from another dataset, rather than copying its data, you must use a provider component. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

Cloning a client dataset cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

CloneCursor takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

Adding application-specific information to the data

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the *SetOptionalParam* method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the *GetOptionalParam* method, passing in the name that was used when the information was stored.

Using a client dataset to cache updates

By default, when you edit data in most datasets, every time you delete or post a record, the dataset generates a transaction, deletes or writes that record to the database server, and commits the transaction. If there is a problem writing changes to the database, your application is notified immediately: the dataset raises an exception when you post the record.

If your dataset uses a remote database server, this approach can degrade performance due to network traffic between your application and the server every time you move to a new record after editing the current record. To minimize the network traffic, you may want to cache updates locally. When you cache updates, you application retrieves data from the database, caches and edits it locally, and then applies the cached updates to the database in a single transaction. When you cache updates, changes to a dataset (such as posting changes or deleting records) are stored locally instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also can't see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

Although the BDE and ADO provide alternate mechanisms for caching updates, using a client dataset for caching updates has several advantages:

- Applying updates when datasets are linked in master/detail relationships is handled for you. This ensures that updates to multiple linked datasets are applied in the correct order.
- Client datasets give you the maximum of control over the update process. You can set properties to influence the SQL that is generated for updating records, specify the table to use when updating records from a multi-table join, or even apply updates manually from a *BeforeUpdateRecord* event handler.
- When errors occur applying cached updates to the database server, only client datasets (and dataset providers) provide you with information about the current record value on the database server in addition to the original (unedited) value from your dataset and the new (edited) value of the update that failed.
- Client datasets let you specify the number of update errors you want to tolerate before the entire update is rolled back.

Overview of using cached updates

To use cached updates, the following order of processes must occur in an application:

1. **Indicate the data you want to edit.** How you do this depends on the type of client dataset you are using:

   - If you are using *TClientDataSet*, Specify the provider component that represent the data you want to edit. This is described in "Specifying a provider" on page 23-24.
   - If you are using a client dataset associated with a particular data access mechanism, you must
     - Identify the database server by setting the *DBConnection* property to an appropriate connection component.
     - Indicate what data you want to see by specifying the *CommandText* and *CommandType* properties. *CommandType* indicates whether *CommandText* is an SQL statement to execute, the name of a stored procedure, or the name of a table. If *CommandText* is a query or stored procedure, use the *Params* property to provide any input parameters.
     - Optionally, use the *Options* property to indicate whether nested detail sets and BLOB data should be included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) should be disabled, whether a single update can affect multiple server records, and whether the client dataset's records are refreshed when it applies updates. *Options* is identical to a provider's *Options* property. As a result, it allows you to set options that are not relevant or appropriate. For example, there is no reason to include *poIncFieldProps*, because the client dataset does not fetch its data from a dataset with persistent fields. Conversely, you do not want to exclude *poAllowCommandText*, which is included by default, because that would disable the *CommandText* property, which the client dataset uses to specify what data it wants. For information on the provider's *Options* property, see "Setting options that influence the data packets" on page 24-5.

2. **Display and edit the data,** permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory. This process is described in "Editing data" on page 23-5.

3. **Fetch additional records as necessary.** By default, client datasets fetch all records and store them in memory. If a dataset contains many records or records with large BLOB fields, you may want to change this so that the client dataset fetches only enough records for display and re-fetches as needed. For details on how to control the record-fetching process, see "Requesting data from the source dataset or document" on page 23-25.

4. **Optionally, refresh the records.** As time passes, other users may modify the data on the database server. This can cause the client dataset's data to deviate more and more from the data on the server, increasing the chance of errors when you apply updates. To mitigate this problem, you can refresh records that have not already been edited. See "Refreshing records" on page 23-29 for details.

5 **Apply the locally cached records to the database** or cancel the updates. For each record written to the database, a *BeforeUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache. For more information about applying updates to the database, see "Updating records" on page 23-19.

Instead of applying updates, an application can cancel the updates, emptying the change log without writing the changes to the database. You can cancel the updates by calling *CancelUpdates* method. All deleted records in the cache are undeleted, modified records revert to original values, and newly inserted record simply disappear.

Choosing the type of dataset for caching updates

Delphi includes some specialized client dataset components for caching updates. Each client dataset is associated with a particular data access mechanism. These are listed in Table 23.3:

Table 23.3 Specialized client datasets for caching updates

Client dataset	Data access mechanism
TBDEClientDataSet	Borland Database Engine
TSQLClientDataSet	dbExpress
TIBClientDataSet	InterBase Express

In addition, you can cache updates using the generic client dataset (*TClientDataSet*) with an external provider and source dataset. For information about using *TClientDataSet* with an external provider, see "Using a client dataset with a provider" on page 23-23.

Note The specialized client datasets associated with each data access mechanism actually use a provider and source dataset as well. However, both the provider and the source dataset are internal to the client dataset.

It is simplest to use one of the specialized client datasets to cache updates. However, there are times when it is preferable to use *TClientDataSet* with an external provider:

- If you are using a data access mechanism that does not have a specialized client dataset, you must use *TClientDataSet* with an external provider component. For example, if the data comes from an XML document or custom dataset.
- If you are working with tables that are related in a master/detail relationship, you should use *TClientDataSet* and connect it, using a provider, to the master table of two source datasets linked in a master/detail relationship. The client dataset sees the detail dataset as a nested dataset field. This approach is necessary so that updates to master and detail tables can be applied in the correct order.

- If you want to code event handlers that respond to the communication between the client dataset and the provider (for example, before and after the client dataset fetches records from the provider), you must use *TClientDataSet* with an external provider component. The specialized client datasets publish the most important events for applying updates (*OnReconcileError*, *BeforeUpdateRecord* and *OnGetTableName*), but do not publish the events surrounding communication between the client dataset and its provider, because they are intended primarily for multi-tiered applications.
- When using the BDE, you may want to use an external provider and source dataset if you need to use an update object. Although it is possible to code an update object from the *BeforeUpdateRecord* event handler of *TBDEClientDataSet*, it can be simpler just to assign the *UpdateObject* property of the source dataset. For information about using update objects, see "Using update objects to update a dataset" on page 20-39.

Indicating what records are modified

While the user edits a client dataset, you may find it useful to provide feedback about the edits that have been made. This is especially useful if you want to allow the user to undo specific edits, for example, by navigating to them and clicking an "Undo" button.

The *UpdateStatus* method and *StatusFilter* properties are useful when providing feedback on what updates have occurred:

- *UpdateStatus* indicates what type of update, if any, has occurred for the current record. It can be any of the following values:
  - *usUnmodified* indicates that the current record is unchanged.
  - *usModified* indicates that the current record has been edited.
  - *usInserted* indicates a record that was inserted by the user.
  - *usDeleted* indicates a record that was deleted by the user.
- *StatusFilter* controls what type of updates in the change log are visible. *StatusFilter* works on cached records in much the same way as filters work on regular data. *StatusFilter* is a set, so it can contain any combination of the following values:
  - *usUnmodified* indicates an unmodified record.
  - *usModified* indicates a modified record.
  - *usInserted* indicates an inserted record.
  - *usDeleted* indicates a deleted record.

  By default, *StatusFilter* is the set [*usModified*, *usInserted*, *usUnmodified*]. You can add *usDeleted* to this set to provide feedback about deleted records as well.

Note *UpdateStatus* and *StatusFilter* are also useful in *BeforeUpdateRecord* and *OnReconcileError* event handlers. For information about *BeforeUpdateRecord*, see "Intervening as updates are applied" on page 23-20. For information about *OnReconcileError*, see "Reconciling update errors" on page 23-22.

The following example shows how to provide feedback about the update status of records using the *UpdateStatus* method. It assumes that you have changed the

StatusFilter property to include *usDeleted*, allowing deleted records to remain visible in the dataset. It further assumes that you have added a calculated field to the dataset called "Status."

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  with ClientDataSet1 do begin
    case UpdateStatus of
      usUnmodified: FieldByName('Status').AsString := '';
      usModified: FieldByName('Status').AsString := 'M';
      usInserted: FieldByName('Status').AsString := 'I';
      usDeleted: FieldByName('Status').AsString := 'D';
    end;
  end;
end;
```

Updating records

The contents of the change log are stored as a data packet in the client dataset's *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database (or source dataset or XML document).

When a client applies updates to the server, the following steps occur:

1. The client application calls the *ApplyUpdates* method of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the (internal or external) provider. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records.
2. The provider applies the updates, caching any problem records that it can't resolve itself. See "Responding to client update requests" on page 24-8 for details on how the provider applies updates.
3. The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client dataset attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

Applying updates

Changes made to the client dataset's local copy of data are not sent to the database server (or XML document) until the client application calls the *ApplyUpdates* method. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the provider. (Note that, when using most client datasets, the provider is internal to the client dataset.)

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log contains all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

1 It indirectly calls the provider's *ApplyUpdates* method. The provider's *ApplyUpdates* method writes the updates to the database, source dataset, or XML document and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.

2 The client dataset 's *ApplyUpdates* method then attempts to reconcile these problem records by calling the *Reconcile* method. *Reconcile* is an error-handling routine that calls the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see "Reconciling update errors" on page 23-22.

3 Finally, *Reconcile* removes successfully applied changes from the change log and updates *Data* to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

Important In some cases, the provider can't determine how to apply updates (for example, when applying updates from a stored procedure or multi-table join). Client datasets and provider components generate events that let you handle these situations. See "Intervening as updates are applied" below for details.

Tip If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. *TClientDataSet* receives a *BeforeApplyUpdates* event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), *TClientDataSet* receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

Intervening as updates are applied

When a client dataset applies its updates, the provider determines how to handle writing the insertions, deletions, and modifications to the database server or source dataset. When you use *TClientDataSet* with an external provider component, you can use the properties and events of that provider to influence the way updates are applied. These are described in "Responding to client update requests" on page 24-8.

When the provider is internal, however, as it is for any client dataset associated with a data access mechanism, you can't set its properties or provide event handlers. As a result, the client dataset publishes one property and two events that let you influence how the internal provider applies updates.

- *UpdateMode* controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see "Influencing how updates are applied" on page 24-9.
- *OnGetTableName* lets you supply the provider with the name of the database table to which it should apply updates. This lets the provider generate the SQL statements for updates when it can't identify the database table from the stored procedure or query specified by *CommandText*. For example, if the query executes a multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates.

  An *OnGetTableName* event handler has three parameters: the internal provider component, the internal dataset that fetched the data from the server, and a parameter to return the table name to use in the generated SQL.
- *BeforeUpdateRecord* occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.)

  A *BeforeUpdateRecord* event handler has five parameters: the internal provider component, the internal dataset that fetched the data from the server, a delta packet that is positioned on the record that is about to be updated, an indication of whether the update is an insertion, deletion, or modification, and a parameter that returns whether the event handler performed the update.The use of these is illustrated in the following event handler. For simplicity, the example assumes the SQL statements are available as global variables that only need field values:

```
procedure TForm1.SQLClientDataSet1BeforeUpdateRecord(Sender: TObject;
   SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
   var Applied Boolean);
var
  SQL: string;
  Connection: TSQLConnection;
begin
  Connection := (SourceDS as TCustomSQLDataSet).SQLConnection;
  case UpdateKind of
  ukModify:
    begin
     { 1st dataset: update Fields[1], use Fields[0] in where clause }
      SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: update Fields[2], use Fields[3] in where clause }
      SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukDelete:
    begin
     { 1st dataset: use Fields[0] in where clause }
      SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: use Fields[3] in where clause }
      SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  ukInsert:
    begin
     { 1st dataset: values in Fields[0] and Fields[1] }
       SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue, DeltaDS.Fields[1].NewValue]);
      Connection.Execute(SQL, nil, nil);
     { 2nd dataset: values in Fields[2] and Fields[3] }
      SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
      Connection.Execute(SQL, nil, nil);
    end;
  end;
  Applied := True;
end;
```

Reconciling update errors

There are two events that let you handle errors that occur during the update process:

- During the update process, the internal provider generates an *OnUpdateError* event every time it encounters an update that it can't handle. If you correct the problem in an *OnUpdateError* event handler, then the error does not count toward the maximum number of errors passed to the *ApplyUpdates* method. This event only occurs for client datasets that use an internal provider. If you are using *TClientDataSet*, you can use the provider component's *OnUpdateError* event instead.
- After the entire update operation is finished, the client dataset generates an *OnReconcileError* event for every record that the provider could not apply to the database server.

You should always code an *OnReconcileError* or *OnUpdateError* event handler, even if only to discard the records returned that could not be applied. The event handlers for these two events work the same way. They include the following parameters:

- *DataSet:* A client dataset that contains the updated record which couldn't be applied. You can use this dataset's methods to get information about the problem record and to edit the record in order to correct any problems. In particular, you will want to use the *CurValue*, *OldValue*, and *NewValue* properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in your event handler.
- *E:* An object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.
- *UpdateKind:* The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).

- *Action:* A **var** parameter that indicates what action to take when the event handler exits. In your event handler, you set this parameter to
  - Skip this record, leaving it in the change log. (rrSkip or raSkip)
  - Stop the entire reconcile operation. (rrAbort or raAbort)
  - Merge the modification that failed into the corresponding record from the server. (rrMerge or raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
  - Replace the current update in the change log with the value of the record in the event handler, which has presumably been corrected. (rrApply or raCorrect)
  - Ignore the error completely. (rrIgnore) This possibility only exists in the *OnUpdateError* event handler, and is intended for the case where the event handler applies the update back to the database server. The updated record is removed from the change log and merged into *Data*, as if the provider had applied the update.
  - Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel) This possibility only exists in the *OnReconcileError* event handler.
  - Update the current record value to match the record on the server. (raRefresh) This possibility only exists in the *OnReconcileError* event handler.

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit which ships in the objrepos directory. (To use this dialog, add RecError to your uses clause.)

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Using a client dataset with a provider

A client dataset uses a provider to supply it with data and apply updates when

- It caches updates from a database server or another dataset.
- It represents the data in an XML document.
- It stores the data in the client portion of a multi-tiered application.

For any client dataset other than *TClientDataSet*, this provider is internal, and so not directly accessible by the application. With *TClientDataSet*, the provider is an external component that links the client dataset to an external source of data.

An external provider component can reside in the same application as the client dataset, or it can be part of a separate application running on another system. For more information about provider components, see Chapter 24, "Using provider components." For more information about applications where the provider is in a separate application on another system, see Chapter 25, "Creating multi-tiered applications."

When using an (internal or external) provider, the client dataset always caches any updates. For information on how this works, see "Using a client dataset to cache updates" on page 23-15.

The following topics describe additional properties and methods of the client dataset that enable it to work with a provider.

Specifying a provider

Unlike the client datasets that are associated with a data access mechanism, *TClientDataSet* has no internal provider component to package data or apply updates. If you want it to represent data from a source dataset or XML document, therefore, you must associated the client dataset with an external provider component.

The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

- If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This works as long as the provider has the same *Owner* as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset's *SetProvider* method

  If you think you may eventually scale up to a remote provider, or if you want to make calls directly to the *IAppServer* interface, you can also set the *RemoteServer* property to a *TLocalConnection* component. If you use *TLocalConnection*, the *TLocalConnection* instance manages the list of all providers that are local to the application, and handles the client dataset's *IAppServer* calls. If you do not use *TLocalConnection*, Delphi creates a hidden object that handles the *IAppServer* calls from the client dataset.

- If the provider is on a remote application server, then, in addition to the *ProviderName* property, you need to specify a component that connects the client dataset to the application server. There are two properties that can handle this task: *RemoteServer*, which specifies the name of a connection component from which to get a list of providers, or *ConnectionBroker*, which specifies a centralized broker that provides an additional level of indirection between the client dataset and the connection component. The connection component and, if used, the connection broker, reside in the same data module as the client dataset. The connection component establishes and maintains a connection to an application server, sometimes called a "data broker". For more information, see "The structure of the client application" on page 25-4.

  At design time, after you specify *RemoteServer* or *ConnectionBroker,* you can select a provider from the drop-down list for the *ProviderName* property in the Object

Inspector. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.

Note If the connection component is an instance of *TDCOMConnection*, the application server must be registered on the client machine.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

Requesting data from the source dataset or document

Client datasets can control how they fetch their data packets from a provider. By default, they retrieve all records from the source dataset. This is true whether the source dataset and provider are internal components (as with *TBDEClientDataSet, TSQLClientDataSet,* and *TIBClientDataSet*), or separate components that supply the data for *TClientDataSet*.

You can change how the client dataset fetches records using the *PacketRecords* and *FetchOnDemand* properties.

Incremental fetching

By changing the *PacketRecords* property, you can specify that the client dataset fetches data in smaller chunks. *PacketRecords* specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
```

This process of fetching records in batches is called “incremental fetching”. Client datasets use incremental fetching when *PacketRecords* is greater than zero.

To fetch each batch of records, the client dataset calls *GetNextPacket*. Newly fetched packets are appended to the end of the data already in the client dataset. *GetNextPacket* returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

Warning Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server. See “Supporting state information in remote data modules” on page 25-19 for information on how to use incremental fetching with stateless remote data modules.

Note You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

Fetch-on-demand

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is *True* (the default), the client dataset automatically fetches records as needed. To prevent automatic fetching of records, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *False*, the application must explicitly call *GetNextPacket* to fetch records.

For example, Applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the server.

The provider controls whether the records in data packets include BLOB data and nested detail datasets. If the provider excludes this information from records, the *FetchOnDemand* property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If *FetchOnDemand* is *False,* and the provider does not include BLOB data and detail datasets with records, you must explicitly call the *FetchBlobs* or *FetchDetails* method to retrieve this information.

Getting parameters from the source dataset

There are two circumstances when the client dataset needs to fetch parameter values:

- The application needs the value of output parameters on a stored procedure.
- The application wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

Client datasets store parameter values in their *Params* property. These values are refreshed with any output parameters when the client dataset fetches data from the source dataset. However, there may be times a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the *FetchParams* method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

Note There is never a need to call *FetchParams* when the client dataset uses an internal provider and source dataset, because the *Params* property always reflects the parameters of the internal source dataset. With *TClientDataSet*, the *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a table-type dataset, there are no parameters to fetch.

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the *Execute* method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

Passing parameters to the source dataset

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Input parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the source dataset at design time or at runtime. At design time, select the client dataset and double-click the *Params* property in the Object Inspector. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the Object Inspector to edit the properties of that parameter.

At runtime, use the *CreateParam* method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code adds an input parameter named CustNo with a value of 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
  AsInteger := 605;
```

If the client dataset is not active, you can send the parameters to the application server and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

Sending query or stored procedure parameters

When the client dataset's *CommandType* property is *ctQuery* or *ctStoredProc*, or, if the client dataset is a *TClientDataSet* instance, when the associated provider represents the results of a query or stored procedure, you can use the *Params* property to specify parameter values. When the client dataset requests data from the source dataset or uses its *Execute* method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

Note Parameter names should match the names of the corresponding parameters on the source dataset.

Limiting records with parameters

If the client dataset is

- a *TClientDataSet* instance whose associated provider represents a *TTable* or *TSQLTable* component
- a *TSQLClientDataSet* or *TBDEClientDataSet* instance whose *CommandType* property is *ctTable*

then it can use the *Params* property to limit the records that it caches in memory. Each parameter represents a field value that must be matched before a record can be included in the client dataset's data. This works much like a filter, except that with a filter, the records are still cached in memory, but unavailable.

Each parameter name must match the name of a field. When using *TClientDataSet*, these are the names of fields in the *TTable* or *TSQLTable* component associated with the provider. When using *TSQLClientDataSet* or *TBDEClientDataSet*, these are the names of fields in the table on the database server. The data in the client dataset then includes only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset, it passes this parameter value. The provider then sends only the records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

Handling constraints from the server

When a database server defines constraints on what data is valid, it is useful if the client dataset knows about them. That way, the client dataset can ensure that user edits never violate those server constraints. As a result, such violations are never passed to the database server where they would be rejected. This means fewer updates generate error conditions during the updating process.

Regardless of the source of data, you can duplicate such server constraints by explicitly adding them to the client dataset. This process is described in “Specifying custom constraints” on page 23-7.

It is more convenient, however, if the server constraints are automatically included in data packets. Then you need not explicitly specify default expressions and constraints, and the client dataset changes the values it enforces when the server constraints change. By default, this is exactly what happens: if the source dataset is aware of server constraints, the provider automatically includes them in data packets and the client dataset enforces them when the user posts edits to the change log.

Note Only datasets that use the BDE can import constraints from the server. This means that server constraints are only included in data packets when using *TBDEClientDataSet* or *TClientDataSet* with a provider that represents a BDE-based dataset. For more information on how to import server constraints and how to prevent a provider from including them in data packets, see “Handling server constraints” on page 24-12.

Note For more information on working with the constraints once they have been imported, see “Using server constraints” on page 19-21.

While importing server constraints and expressions is an extremely valuable feature that helps an application preserve data integrity, there may be times when it needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value of a field, but the client dataset uses incremental fetching, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call the *DisableConstraints* method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset’s *EnableConstraints* method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

Tip Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

Refreshing records

Client datasets work with an in-memory snapshot of the data from the source dataset. If the source dataset represents server data, then as time elapses other users may modify that data. The data in the client dataset becomes a less accurate picture of the underlying data.

Like any other dataset, client datasets have a *Refresh* method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client datasets can also update the data while leaving the change log intact. To do this, call the *RefreshRecord* method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

Warning It is not always appropriate to call *RefreshRecord*. If the user’s edits conflict with changes made to the underlying dataset by other users, calling *RefreshRecord* masks this conflict. When the client dataset applies its updates, no reconcile error occurs and the application can’t resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following *AfterScroll* refreshes the current record every time the user moves to a new record (ensuring the most up-to-date value), but only when it is safe to do so.:

```
procedure TForm1.ClientDataSet1AfterScroll(DataSet: TDataSet);
begin
  if ClientDataSet1.UpdateStatus = usUnModified then
    ClientDataSet1.RefreshRecord;
end;
```

Communicating with providers using custom events

Client datasets communicate with a provider component through a special interface called *IAppServer*. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server

TClientDataSet provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1. The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
2. The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.
3. The provider goes through its normal process of assembling a data packet (including all the accompanying events).
4. The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.
5. The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between client dataset and provider.

In addition, the client dataset has a special method, *DataRequest*, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an is the *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

Overriding the source dataset

The client datasets that are associated with a particular data access mechanism use the *CommandText* and *CommandType* properties to specify the data they represent. When using *TClientDataSet*, however, the data is specified by the source dataset, not the client dataset. Typically, this source dataset has a property that specifies an SQL statement to generate the data or the name of a database table or stored procedure.

If the provider allows, *TClientDataSet* can override the property on the source dataset that indicates what data it represents. That is, if the provider permits, the client dataset's *CommandText* property replaces the property on the provider's dataset that specifies what data it represents. This allows *TClientDataSet* to specify dynamically what data it wants to see.

By default, external provider components do not let client datasets use the *CommandText* value in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider. Otherwise, the value of *CommandText* is ignored.

Note Never remove *poAllowCommandText* from the *Options* property of *TSQLClientDataSet*, *TBDEClientDataSet*, or *TIBClientDataSet*. The client dataset's *Options* property is forwarded to the internal provider, so removing *poAllowCommandText* prevents the client dataset from specifying what data to access.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the *AppServer* property. This property represents the interface through which the client dataset communicates with its provider.

Using a client dataset with file-based data

Client datasets can work with dedicated files on disk as well as server data. This allows them to be used in file-based database applications and "briefcase model" applications. The special files that client datasets use for their data are called MyBase.

Tip All client datasets are appropriate for a briefcase model application, but for a pure MyBase application (one that does not use a provider), it is preferable to use *TClientDataSet*, because it involves less overhead.

In a pure MyBase application, the client application cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables

- Load saved data
- Merge edits into its data
- Save data

Creating a new dataset

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset using persistent fields or field and index definitions. This follows the same scheme as creating any table-type dataset. See "Creating and deleting tables" on page 18-37 for details.
- You can copy an existing dataset (at design or runtime). See "Copying data from another dataset" on page 23-13 for more information about copying existing datasets.
- You can create a client dataset from an arbitrary XML document. See "Converting XML documents into data packets" on page 26-6 for details.

Once the dataset is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you start with the metadata for your client dataset already defined, making it easier to set up the user interface.

Loading data from a file or stream

To load data from a file, call a client dataset's *LoadFromFile* method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the *FileName* property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile (SaveToStream)* method, or generated from an XML document. For more information about saving data to a file or stream, see "Saving data to a file or stream" on page 23-33. For information about creating client dataset data from an XML document, see Chapter 26, "Using XML in database applications."

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property. However, the only indexes that are read from the file are those that were created with the dataset.

Merging changes into data

When you edit the data in a client dataset, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the *MergeChangeLog* method. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, *Data* contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the *ChangeCount* property to *0*.

Warning Do not call *MergeChangeLog* for client datasets that use a provider. In this case, call *ApplyUpdates* to write changes to the database. For more information, see “Applying updates” on page 23-19.

Note It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see “Assigning data directly” on page 23-13.

If you do not want to use the extended undo capabilities of the change log, you can set the client dataset’s *LogChanges* property to *False*. When *LogChanges* is *False*, edits are automatically merged when you post records and there is no need to call *MergeChangeLog*.

Saving data to a file or stream

Even when you have merged changes into the data of a client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the *SaveToFile* method.

SaveToFile takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

Note *SaveToFile* does not preserve any indexes you added to the client dataset at runtime, only indexes that were added when you created the client dataset.

If you always save the data to the same file, you can use the *FileName* property instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the *SaveToStream* method. *SaveToStream* takes one parameter, a stream object that receives the data.

Note If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component on the application server.

Chapter

24

Using provider components

Provider components (*TDataSetProvider* and *TXMLTransformProvider*) supply the most common mechanism by which client datasets obtain their data. Providers

- Receive data requests from a client dataset (or XML broker), fetch the requested data, package the data into a transportable data packet, and return the data to the client dataset (or XML broker). This activity is called “providing.”
- Receive updated data from a client dataset (or XML broker), apply updates to the database server, source dataset, or source XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called “resolving.”

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in a dataset or XML document or to apply updates. However, provider components include a number of events and properties that allow your application more direct control over what information is packaged for clients and how your application responds to client requests.

When using *TBDEClientDataSet*, *TSQLClientDataSet*, or *TIBClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet* or *TXMLBroker*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. The client datasets that have internal providers surface some of the internal provider’s properties and events as their own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset (or XML broker), or it can reside on an application server as part of a multi-tiered application.

This chapter describes how to use a provider component to control the interaction with client datasets or XML brokers.

Determining the source of data

When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. Depending on your version of Delphi, you can specify the source as one of the following:

- To provide the data from a dataset, use *TDataSetProvider*.
- To provide the data from an XML document, use *TXMLTransformProvider*.

Using a dataset as the source of the data

If the provider is a dataset provider (*TDataSetProvider*), set the *DataSet* property of the provider to indicate the source dataset. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

TDataSetProvider interacts with the source dataset using the *IProviderSupport* interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don’t do anything or that raise exceptions.

The dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, *dbExpress* datasets, and InterBase Express datasets) override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets don’t add anything to the inherited *IProviderSupport* implementation, but can still be used as a source dataset as long as the *ResolveToDataSet* property of the provider is *True*.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider only provides data packets on a read-only basis (that is, if it does not apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

Using an XML document as the source of the data

If the provider is an XML provider, set the *XMLDataFile* property of the provider indicate the source document.

XML providers must transform the source document into data packets, so in addition to indicating the source document, you must also specify how to transform that document into data packets. This transformation is handled by the provider’s *TransformRead* property. *TransformRead* represents a *TXMLTransform* object. You can set its properties to specify what transformation to use, and use its events to provide your own input to the transformation. For more information on using XML providers, see “Using an XML document as the source for a provider” on page 26-8.

Communicating with the client dataset

All communication between a provider and a client dataset or XML broker takes place through an *IAppServer* interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you, or by a *TLocalConnection* component. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module.

Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset or XML broker. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of a client dataset.

Table 24.1 lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an OleVariant parameter called *OwnerData* that allows a client dataset and a provider to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

Table 24.1 AppServer interface members

IAppServer	provider component	TClientDataSet
AS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event.
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method.
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event.
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetParams event, AfterGetParams event.
AS_GetProviderNames method	Used to identify all available providers.	Used to create a design-time list for ProviderName property.
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event

Choosing how to apply updates using a dataset provider

TXMLTransformProvider components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server).

However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you use may not support the use of SQL statements (for example if you are providing from a *TClientDataSet* component).

TDataSetProvider lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the *ResolveToDataSet* property. When this property is *True*, updates are applied to the dataset. When it is *False*, updates are applied directly to the underlying database server.

Controlling what information is included in data packets

When working with a dataset provider, there are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets
- Setting options that influence the data packets
- Adding custom information to data packets

Note These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.

Specifying what fields appear in data packets

When using a dataset provider, you can control what fields are included in data packets by creating persistent fields on the dataset that the provider uses to build data packets. The provider then includes only these fields. Fields whose values are generated dynamically by the source dataset (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields. For information about persistent fields, see “Persistent field components” on page 19-3.

If the client dataset will be editing the data and applying updates, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields provided only to ensure uniqueness, set the *ProviderFlags* property for those fields to include *pfHidden*.

Note Including enough fields to avoid duplicate records is also a consideration when the provider's source dataset represents a query. You must specify the query so that it includes enough fields to ensure all records are unique, even if your application does not use all the fields.

Setting options that influence the data packets

The *Options* property of a dataset provider lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

Table 24.2 Provider options

Value	Meaning
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poReadOnly	The client dataset can't apply updates to the provider.
poDisableEdits	Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises exception. (This does not affect the client dataset's ability to insert or delete records).
poDisableInserts	Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data)
poDisableDeletes	Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records)
poFetchBlobsOnDemand	BLOB field values are not included in data packets. Instead, client datasets must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchBlobs* method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider's dataset represents the master of a master/detail relationship, nested detail values are not included in data packets. Instead, client datasets request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchDetails* method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues.*

Table 24.2 Provider options (continued)

Value	Meaning
poCascadeDeletes	When the provider's dataset represents the master of a master/ detail relationship, the server automatically deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.
poCascadeUpdates	When the provider's dataset represents the master of a master/ detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poNoReset	Client datasets can't specify that the provider should reposition the cursor on the first record before providing data.
poPropogateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client can override the associated dataset's SQL text or the name of the table or stored procedure it represents.
poRetainServerOrder	The client dataset should not re-sort the records in the dataset to enforce a default order.

Adding custom information to data packets

Dataset providers can add application-defined information to data packets using the *OnGetDataSetProperties* event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their *GetOptionalParam* method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client dataset may never be aware of the information, but the provider can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Add multiple attributes by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only the time the data was provided is returned when client datasets apply updates:

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

When the client dataset applies updates, the time the original records were provided can be read in the provider's *OnUpdateData* event:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

Responding to client data requests

Usually client requests for data are handled automatically. A client dataset or XML broker requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset or XML document, creating a data packet, and sending the packet to the client.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client, write an *OnGetData* event handler. *OnGetData* event handlers provide the data packet as a parameter in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client.

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers. For a discussion of persistent state information in application servers, see "Supporting state information in remote data modules" on page 25-19.

Responding to client update requests

A provider applies updates to database records based on a *Delta* data packet received from a client dataset or XML broker. The client requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information with a client dataset before and after the call to *ApplyUpdates*. This communication takes place using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers. For a discussion of persistent state information in application servers, see "Supporting state information in remote data modules" on page 25-19.

If you are using a dataset provider, a number of additional events allow you more control:

When a dataset provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdateData* event, the provider writes the changes to the database or source dataset.

The provider performs the update on a record-by-record basis. Before the dataset provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error. Usually errors occur because the change violates a server constraint or a database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset's request to apply updates.

Update errors can be processed by either the dataset provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won't be using. To filter the client dataset on the update status of its records, set its *StatusFilter* property.

Note Applications must supply extra support when the updates are directed at a dataset that does not represent a single table. For details on how to do this, see "Applying updates to datasets that do not represent a single table" on page 24-11.

Editing delta packets before updating the database

Before a dataset provider applies updates to the database, it generates an *OnUpdateData* event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the *UpdateStatus* property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in Table 24.3.

Table 24.3 UpdateStatus values

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

Influencing how updates are applied

The *OnUpdateData* event also gives your dataset provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the *UpdateMode* property of the provider. *UpdateMode* can be assigned any of the following values:

Table 24.4 UpdateMode values

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereKeyOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the *ProviderFlags* property. *ProviderFlags* is a set that can include any of the values in Table 24.5

Table 24.5 ProviderFlags values

Value	Description
pfInWhere	The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when *UpdateMode* is *upWhereAll* or *upWhereChanged*.
pfInUpdate	The field appears in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of generated statements when *UpdateMode* is *upWhereKeyOnly*.
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

Note You can use the *UpdateFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

Screening individual updates

Immediately before each update is applied, a dataset provider receives a *BeforeUpdateRecord* event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

Resolving update errors on the provider

When an error condition arises as the dataset provider tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the *NewValue*, *OldValue*, and *CurValue* properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

Applying updates to datasets that do not represent a single table

When a dataset provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as table-type datasets or “live” *TQuery* components. Automatic updates are a problem however, if the provider must apply updates to the data underlying a stored procedure with a result set or a multi-table query. There is no easy way to obtain the name of the table to which updates should be applied.

If the query or stored procedure is a BDE-enabled dataset (*TQuery* or *TStoredProc*) and it has an associated update object, the provider uses the update object. However, if there is no update object, you can supply the table name programmatically in an *OnGetTableName* event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the *BeforeUpdateRecord* event of the provider. Once this event handler has applied an update, you can set the event handler's *Applied* parameter to *True* so that the provider does not generate an error.

Note If the provider is associated with a BDE-enabled dataset, you can use an update object in the *BeforeUpdateRecord* event handler to apply updates using customized SQL statements. See "Using update objects to update a dataset" on page 20-39 for details.

Responding to client-generated events

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the *OnDataRequest* event.

OnDataRequest is not part of the normal functioning of the provider. It is simply a hook to allow your client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, the client application calls the *DataRequest* method of the client dataset.

Handling server constraints

Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints:

- NOT NULL, to guarantee that a value supplied to a column has a value.
- NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record.
- CHECK, to guarantee that a value supplied to a column falls within a certain range, or is one of a limited number of possible values.
- CONSTRAINT, a table-wide check constraint that applies to multiple columns.

- PRIMARY KEY, to designate one or more columns as the table's primary key for indexing purposes.
- FOREIGN KEY, to designate one or more columns in a table that reference another table.

Note This list is not exclusive. Your database server may support some or all of these constraints in part or in whole, and may support additional constraints. For more information about supported constraints, see your server documentation.

Database server constraints obviously duplicate many kinds of data checks that traditional desktop database applications manage. You can take advantage of server constraints in multi-tiered database applications without having to duplicate the constraints in application server or client application code.

If the provider is working with a BDE-enabled dataset, the *Constraints* property lets you replicate and apply server constraints to data passed to and received from client datasets. When *Constraints* is *True* (the default), server constraints stored in the source dataset are included in data packets and affect client attempts to update data.

Important Before the provider can pass constraint information on to client datasets, it must retrieve the constraints from the database server. To import database constraints from the server, use SQL Explorer to import the database server's constraints and default expressions into the Data Dictionary. Constraints and default expressions in the Data Dictionary are automatically made available to BDE-enabled datasets.

There may be times when you do not want to apply server constraints to data sent to a client dataset. For example, a client dataset that receives data in packets and permits local updating of records prior to fetching more records may need to disable some server constraints that might be triggered because of the temporarily incomplete set of data. To prevent constraint replication from the provider to a client dataset, set *Constraints* to *False*. Note that client datasets can disable and enable constraints using the *DisableConstraints* and *EnableConstraints* methods. For more information about enabling and disabling constraints from the client dataset, see "Handling constraints from the server" on page 23-28.

Chapter

25

Creating multi-tiered applications

This chapter describes how to create a multi-tiered, client/server database application. A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the “three-tiered model,” a multi-tiered application is partitioned into thirds:

- **Client application**: provides a user interface on the user’s machine.
- **Application server**: resides in a central networking location accessible to all clients and provides common data services.
- **Remote database server**: provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a “data broker.” With Delphi you usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Delphi’s support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. This chapter focuses on creating a three-tiered database application. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

Advantages of the multi-tiered database model

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications**. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the Borland Database Engine and the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- **Distributed data processing**. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security**. You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTP, CORBA, or COM+, you can take advantage of the security models they support.

Understanding provider-based multi-tiered applications

Delphi's support for multi-tiered applications use the components on the DataSnap page and the Data Access page of the component palette, plus a remote data module that is created by a wizard on the Multitier page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets.

The components needed for a multi-tiered application are described in Table 25.1:

Table 25.1 Components used in multi-tiered applications

Component	Description
Remote data modules	Specialized data modules that can act as a COM Automation server, SOAP server, or CORBA server to give client applications access to any providers they contain. Used on the application server.
Provider component	A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
Client dataset component	A specialized dataset that uses midas.dll or midaslib.dcu to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server.
Connection components	A family of components that locate the server, form connections, and make the *IAppServer* interface available to client datasets. Each connection component is specialized to use a particular communications protocol.

The provider and client dataset components require midas.dll or midaslib.dcu, which manages datasets stored as data packets. (Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using midas.dll, you must deploy it on both application server and client application.)

If you are using BDE-enabled datasets, the application server may also require SQL Explorer to help in database administration and to import server constraints into the Data Dictionary so that they can be checked at any level of the multi-tiered application.

Note You must purchase server licenses for deploying your application server.

An overview of the architecture into which these components fit is described in “Using a multi-tiered architecture” on page 14-12.

Overview of a three-tiered application

The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application:

1. A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface from the application server.
2. The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).
3. The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, field display characteristics) can be included in the metadata of the data packet. This process of packaging data into data packets is called “providing.”

4 The client decodes the data packet and displays the data to the user.

5 As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.

6 Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.

7 The application server decodes the package and posts updates (in the context of a transaction if appropriate). If a record can't be posted (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client's changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called "resolving."

8 When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.

9 The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.

10 The client refreshes its data from the server.

The structure of the client application

To the end user, the client application of a multi-tiered application looks and behaves no differently than a traditional two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a *TClientDataSet* component. For detailed information about using the properties, events, and methods of client datasets, see Chapter 23, "Using client datasets."

TClientDataSet fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Chapter 24, "Using provider components". For details about client dataset features that facilitate its communication with a provider, see "Using a client dataset with a provider" on page 23-23

The client dataset communicates with the provider through the *IAppServer* interface. It gets this interface from a connection component. The connection component establishes the connection to the application server. Different connection components are available for using different communications protocols. These connection components are summarized in the following table:

Table 25.2 Connection components

Component	Protocol
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP (HTTP and XML)
TCorbaConnection	CORBA (IIOP)

Note Delphi also includes a connection component that does not connect to an application server at all, but instead supplies an *IAppServer* interface for client datasets to use when communicating with providers in the same application. This component, *TLocalConnection*, is not required, but makes it easier to later scale up to a multi-tiered application.

For more information about using connection components, see “Connecting to the application server” on page 25-23.

The structure of the application server

When you set up and run an application server, it does not establish any connection with client applications. Instead, connection is initiated and maintained by client applications. The client application uses its connection component to establish a connection to the application server, which it uses to communicate with its selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

The basis of an application server is a remote data module, which is a specialized data module that supports the *IAppServer* interface. Client applications use the *IAppServer* interface to communicate with providers on the application server.

There are four types of remote data modules:

- **TRemoteDataModule**: This is a dual-interface Automation server. Use this type of remote data module if clients use DCOM, HTTP, sockets, or OLE to connect to the application server, unless you want to install the application server with MTS.
- **TMTSDataModule**: This is a dual-interface Automation server. Use this type of remote data module if you are creating the application server as an Active Library (.DLL) that is installed with MTS or COM+. You can use MTS remote data modules with DCOM, HTTP, sockets, or OLE.
- **TCorbaDataModule**: This is a CORBA server. Use this type of remote data module to provide data to CORBA clients.
- **TSoapDataModule**: This is a data module that implements an *IAppServer* descendant as an invokable interface in a Web Service application. Use this type of remote data module to provide data to clients that access data as a Web Service.

If the application server is to be deployed under MTS or COM+, the remote data module includes events for when the application server is activated or deactivated. This allows it to acquire database connections when activated and release them when deactivated.

The contents of the remote data module

As with any data module, you can include any nonvisual component in the remote data module. There are certain components, however, that you must include:

- If the remote data module is exposing information from a database server, it must include a dataset component to represent the records from that database server. Other components, such as a database connection component of some type, may be required to allow the dataset to interact with a database server. For information about datasets, see Chapter 18, “Understanding datasets.” For information about database connection components, see Chapter 17, “Connecting to databases.”

  For every dataset that the remote data module exposes to clients, it must include a dataset provider. A dataset provider packages data into data packets that are sent to client datasets and applies updates received from client datasets back to a source dataset or a database server. For more information about dataset providers, see Chapter 24, “Using provider components.”
- For every XML document that the remote data module exposes to clients, it must include an XML provider. An XML provider acts like a dataset provider, except that it fetches data from and applies updates to an XML document rather than a database server. For more information about XML providers, see “Using an XML document as the source for a provider” on page 26-8.

Note Do not confuse database connection components, which connect datasets to a database server, with the connection components used by client applications in a multi-tiered application. The connection components in multi-tiered applications can be found on the DataSnap page of the Component palette.

Using transactional data modules

You can write an application server that takes advantage of special services for distributed applications that are supplied by MTS (before Windows 2000) or COM+ (under Windows 2000 and later). To do so, you create a transactional data module instead of an ordinary remote data module.

When you use a transactional data module, your application can take advantage of the following special services:

- **Security.** MTS and COM+ provide role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module’s interface. The MTS data module implements the *IsCallerInRole* method, which you lets you check the role of the currently connected client and conditionally allow certain functions based on that role. For more information about MTS and COM+ security, see “Role-based security” on page 39-14.
- **Database handle pooling.** Transactional data modules automatically pool database connections that are made via ADO or (if you are using MTS and turn on MTS POOLING) the BDE. When one client is finished with a database connection,

another client can reuse it. This cuts down on network traffic, because your middle tier does not need to log off of the remote database server and then log on again. When pooling database handles, your database connection component should set its *KeepConnection* property to *False*, so that your application maximizes the sharing of connections. For more information about pooling database handles, see "Database resource dispensers" on page 39-5.

- **Transactions**. When using a transactional data module, you can provide enhanced transaction support beyond that available with a single database connection. Transactional data modules can participate in transactions that span multiple databases, or include functions that do not involve databases at all. For more information about the transaction support provided by transactional objects such as transactional data modules, see "Managing transactions in multi-tiered applications" on page 25-18.
- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your server so that remote data module instances are activated and deactivated on an as-needed basis. When using just-in-time activation and as-soon-as-possible deactivation, your remote data module is instantiated only when it is needed to handle client requests. This prevents it from tying up resources such as database handles when they are not in use.

  Using just-in-time activation and as-soon-as-possible deactivation provides a middle ground between routing all clients through a single remote data module instance, and creating a separate instance for every client connection. With a single remote data module instance, the application server must handle all database calls through a single database connection. This acts as a bottleneck, and can impact performance when there are many clients. With multiple instances of the remote data module, each instance can maintain a separate database connection, thereby avoiding the need to serialize database access. However, this monopolizes resources because other clients can't use the database connection while it is associated with another client's remote data module.

To take advantage of transactions, just-in-time activation, and as-soon-as-possible deactivation, remote data module instances must be stateless. This means you must provide additional support if your client relies on state information. For example, the client must pass information about the current record when performing incremental fetches. For more information about state information and remote data modules in multi-tiered applications, see "Supporting state information in remote data modules" on page 25-19.

By default, all automatically generated calls to a transactional data module are transactional (that is, they assume that when the call exits, the data module can be deactivated and any current transactions committed or rolled back). You can write a transactional data module that depends on persistent state information by setting the *AutoComplete* property to *False*, but it will not support transactions, just-in-time activation, or as-soon-as-possible deactivation unless you use a custom interface.

Warning Application servers containing transactional data modules should not open database connections until the data module is activated. While developing your application, be sure that all datasets are not active and the database is not connected before running your application. In the application itself, add code to open database connections when the data module is activated and close them when it is deactivated.

Pooling remote data modules

Object pooling allows you to create a cache of remote data modules that are shared by their clients, thereby conserving resources. How this works depends on the type of remote data module and on the connection protocol.

If you are creating a transactional data module that will be installed to COM+, you can use the COM+ Component Manager to install the application server as a pooled object. See "Object pooling" on page 39-8 for details.

Even if you are not using a transactional data module, you can take advantage of object pooling if the connection is formed using HTTP (*TWebConnection*). Under this second type of object pooling, you limit the number of instances of your remote data module that are created. This limits the number of database connections that you must hold, as well as any other resources used by the remote data module.

When the Web Server application (which passes calls to your remote data module) receives client requests, it passes them on to the first available remote data module in the pool. If there is no available remote data module, it creates a new one (up to a maximum number that you specify). This provides a middle ground between routing all clients through a single remote data module instance (which can act as a bottleneck), and creating a separate instance for every client connection (which can consume many resources).

If a remote data module instance in the pool does not receive any client requests for a while, it is automatically freed. This prevents the pool from monopolizing resources unless they are used.

To set up object pooling when using a Web connection (HTTP), your remote data module must override the *UpdateRegistry* method. In the overridden method, call *RegisterPooled* when the remote data module registers and *UnregisterPooled* when the remote data module unregisters. When using either method of object pooling, your remote data module must be stateless. This is because a single instance potentially handles requests from several clients. If it relied on persistent state information, clients could interfere with each other. See "Supporting state information in remote data modules" on page 25-19 for more information on how to ensure that your remote data module is stateless.

Choosing a connection protocol

Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future development plans.

Using DCOM connections

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server. However, because DCOM is not included with Windows 95, some older client machines may not have DCOM installed.

DCOM provides the only approach that lets you use security services when writing a transactional data module. These security services are based on assigning roles to the callers of transactional objects. When using DCOM, DCOM identifies the caller to the system that calls your application server (MTS or COM+). Therefore, it is possible to accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes COM calls into the application server on behalf of the client. Because of this, it is impossible to assign roles to separate clients: The runtime executable is, effectively, the only client. For more information about security and transactional objects, see “Role-based security” on page 39-14.

Using Socket connections

TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application, you can’t be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about sockets, see Chapter 32, “Working with sockets.”

Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe on the server are responsible for marshaling *IAppServer* calls.

Note ScktSrvr.exe can run as an NT service application. Register it with the Service manager by starting it using the -install command line option. You can unregister it using the -uninstall command line option.

Before you can use a socket connection, the application server must register its availability to clients using a socket connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableSocketTransport* in the *UpdateRegistry* method. You can remove this call to prevent socket connections to your application server.

Note Because older servers did not add this registration, you can disable the check for whether an application server is registered by unchecking the Connections | Registered Objects Only menu item on ScktSrvr.exe.

When using sockets, there is no protection on the server against client systems failing before they release a reference to interfaces on the application server. While this results in less message traffic than when using DCOM (which sends periodic keep-alive messages), this can result in an application server that can’t release its resources because it is unaware that the client has gone away.

Using Web connections

HTTP lets you create clients that can communicate with an application server that is protected by a firewall. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Chapter 27, “Creating Internet applications.”

Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) that accepts client requests and instantiates the remote data module using COM. Because of this, they are also called Web connections. The connection component on the client and httpsrvr.dll on the server are responsible for marshaling *IAppServer* calls.

Web connections can take advantage of the SSL security provided by wininet.dll (a library of Internet utilities that runs on the client system). Once you have configured the Web server on the server system to require authentication, you can specify the user name and password using the properties of the Web connection component.

As an additional security measure, the application server must register its availability to clients using a Web connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableWebTransport* in the *UpdateRegistry* method. You can remove this call to prevent Web connections to your application server.

Web connections can take advantage of object pooling. This allows your server to create a limited pool of remote data module instances that are available for client requests. By pooling the remote data modules, your server does not consume the resources for the data module and its database connection except when they are needed. For more information on object pooling, see “Pooling remote data modules” on page 25-8.

Unlike most other connection components, you can’t use callbacks when the connection is formed via HTTP.

Using SOAP connections

SOAP is the protocol that underlies Delphi’s support for Web Service applications. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol.

SOAP connections have the advantage that they work in cross-platform applications because they are supported on both the Windows and Linux. Because SOAP connections use HTTP, they have the same advantages as Web connections: HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a “firewall”. For more information about using SOAP to distribute applications in Delphi, see Chapter 31, “Using Web Services.”

As with HTTP connections, you can’t use callbacks when the connection is formed via SOAP. SOAP connections also limit you to a single remote data module in the application server.

Using CORBA connections

CORBA lets you integrate your multi-tiered database applications into an environment that is standardized on CORBA. For example, when working with a client application written in Java, only the CORBA connection is available. Because CORBA (and Java) is available on multiple platforms, this allows you to write cross-platform multi-tiered applications.

By using CORBA, your application automatically gets the benefits of load-balancing, location transparency, and fail-over from the ORB runtime software. In addition, you can add hooks to take advantage of other CORBA services.

Building a multi-tiered application

The general steps for creating a multi-tiered database application are

1. Create the application server.
2. Register or install the application server.
3. Create a client application.

The order of creation is important. You should create and run the application server before you create a client. At design time, you can then connect to the application server to test your client. You can, of course, create a client without specifying the application server at design time, and only supply the server name at runtime. However, doing so prevents you from seeing if your application works as expected when you code at design time, and you will not be able to choose servers and providers using the Object Inspector.

Note If you are not creating the client application on the same system as the server, and you are using a DCOM connection, you may want to register the application server on the client system. This makes the connection component aware of the application server at design time so that you can choose server names and provider names from a drop-down list in the Object Inspector. (If you are using a Web connection, SOAP connection, or socket connection, the connection component fetches the names of registered servers from the server machine.)

Creating the application server

You create an application server very much as you create most database applications. The major difference is that the application server uses a remote data module.

To create an application server, follow these steps:

1. Start a new project:
   - If you are using SOAP as a transport protocol, this should be a new Web Service application. Choose File | New | Other, and on the Web Services page of the new items dialog, choose Web Service application.
   - For any other transport protocol, you need only choose File | New | Application.

   Save the new project.
2. Add a new remote data module to the project. From the main menu, choose File | New | Other, and on the Multitier page of the new items dialog, select
   - **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, or sockets.

- **Transactional Data Module** if you are creating a remote data module that runs under MTS or COM+. Connections can be formed using DCOM, HTTP, or sockets. However, only DCOM supports the security services.
- **CORBA Data Module** if you are creating a CORBA server.
- **SOAP Data Module** if you are creating a SOAP server in a Web Service application.

For more detailed information about setting up a remote data module, see "Setting up the remote data module" on page 25-13.

Note Remote data modules are more than simple data modules. The CORBA data module acts as a CORBA server. The SOAP data module implements an invokable interface in a Web Service application. Other data modules are COM Automation objects.

3 Place the appropriate dataset components on the data module and set them up to access the database server.

4 Place a *TDataSetProvider* component on the data module for each dataset. This provider is required for brokering client requests and packaging data. Set the *DataSet* property for each provider component to the name of the dataset to access. You can set additional properties for the provider. See Chapter 24, "Using provider components" for more detailed information about setting up a provider.

If you are working with data from XML documents, you can use a *TXMLTransformProvider* component instead of a dataset and *TDataSetProvider* component. When using *TXMLTransformProvider*, set the *XMLDataFile* property to specify the XML document from which data is provided and to which updates are applied.

5 Write application server code to implement events, shared business rules, shared data validation, and shared security. When writing this code, you may want to

- Extend the application server's interface to provide additional ways for the client application to call the server. Extending the application server's interface is described in "Extending the application server's interface" on page 25-16.
- Provide transaction support beyond the transactions automatically created when applying updates. Transaction support in multi-tiered database applications is described in "Managing transactions in multi-tiered applications" on page 25-18.
- Create master/detail relationships between the datasets in your application server. Master/detail relationships are described in "Supporting master/detail relationships" on page 25-19.
- Ensure your application server is stateless. Handling state information is described in "Supporting state information in remote data modules" on page 25-19.
- Divide your application server into multiple remote data modules. Using multiple remote data modules is described in "Using multiple remote data modules" on page 25-21.

6 Save, compile, and register or install the application server. Registering an application server is described in “Registering the application server” on page 25-22.

7 If your server application does not use DCOM or SOAP, you must install the runtime software that receives client messages, instantiates the remote data module, and marshals interface calls.

   - For TCP/IP sockets this is a socket dispatcher application, Scktsrvr.exe.
   - For HTTP connections this is httpsrvr.dll, an ISAPI/NSAPI DLL that must be installed with your Web server.
   - For CORBA, this is the VisiBroker ORB.

Setting up the remote data module

When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See “The structure of the application server” on page 25-5 for information on what type of remote data module you need.

Configuring TRemoteDataModule

To add a *TRemoteDataModule* component to your application, choose File | New | Other and select Remote Data Module from the Multitier page of the new items dialog. You will see the Remote Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TRemoteDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TRemoteDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note You can add your own properties and methods to the new interface. For more information, see “Extending the application server’s interface” on page 25-16.

You must specify the threading model in the Remote Data Module wizard. You can choose Single-threaded, Apartment-threaded, Free-threaded, or Both.

- If you choose Single-threaded, COM ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment-threaded, COM ensures that any instance of your remote data module services one request at a time. When writing code in an Apartment-threaded library, you must guard against thread conflicts if you use global variables or objects not contained in the remote data module. This is the recommended model if you are using BDE-enabled datasets. (Note that you will need a session component with its *AutoSessionName* property set to *True* to handle threading issues on BDE-enabled datasets).

- If you choose Free-threaded, your application can receive simultaneous client requests on several threads. You are responsible for ensuring your application is thread-safe. Because multiple clients can access your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables. This is the recommended model if you are using ADO datasets.
- If you choose Both, your library works the same as when you choose Free-threaded, with one exception: all callbacks (calls to client interfaces) are serialized for you.
- If you choose Neutral, the remote data module can receive simultaneous calls on separate threads, as in the Free-threaded model, but COM guarantees that no two threads access the same method at the same time.

If you are creating an EXE, you must also specify what type of instancing to use. You can choose Single instance or Multiple instance (Internal instancing applies only if the client code is part of the same process space.)

- If you choose Single instance, each client connection launches its own instance of the executable. That process instantiates a single instance of the remote data module, which is dedicated to the client connection.
- If you choose Multiple instance, a single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

Configuring TMTSDataModule

To add a *TMTSDataModule* component to your application, choose File | New | Other and select Transactional Data Module from the Multitier page of the new items dialog. You will see the Transactional Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note You can add your own properties and methods to your new interface. For more information, see “Extending the application server’s interface” on page 25-16.

You must specify the threading model in the Transactional Data Module wizard. Choose Single, Apartment, or Both.

- If you choose Single, client requests are serialized so that your application services only one at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment, the system ensures that any instance of your remote data module services one request at a time, and calls always use the same thread. You must guard against thread conflicts if you use global variables or objects not contained in the remote data module. Instead of using global variables, you can use the shared property manager. For more information on the shared property manager, see “Shared property manager” on page 39-6.

- If you choose Both, MTS calls into the remote data module's interface in the same way as when you choose Apartment. However, any callbacks you make to client applications are serialized, so that you don't need to worry about them interfering with each other.

Note The Apartment model under MTS or COM+ is different from the corresponding model under DCOM.

You must also specify the transaction attributes of your remote data module. You can choose from the following options:

- Requires a transaction. When you select this option, every time a client uses your remote data module's interface, that call is executed in the context of a transaction. If the caller supplies a transaction, a new transaction need not be created.
- Requires a new transaction. When you select this option, every time a client uses your remote data module's interface, a new transaction is automatically created for that call.
- Supports transactions. When you select this option, your remote data module can be used in the context of a transaction, but the caller must supply the transaction when it invokes the interface.
- Does not support transactions. When you select this option, your remote data module can't be used in the context of transactions.

Configuring TSoapDataModule

To add a *TSoapDataModule* component to your application, choose File | New | Other and select SOAP Data Module from the Multitier page of the new items dialog. The SOAP data module wizard appears.

You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendant that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TSoapDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

You may want to edit the definitions of the generated interface and *TSoapDataModule* descendant, adding your own properties and methods. These properties and methods are not called automatically, but client applications that request your new interface by name can use any of the properties and methods that you add.

Note To use *TSoapDataModule*, the new data module should be added to a Web Service application. The *IAppServer* interface is an invokable interface, which is registered in the initialization section of the new unit. This allows the invoker component in the main Web module to forward all incoming calls to your data module.

Configuring TCorbaDataModule

To add a *TCorbaDataModule* component to your application, choose File | New and select CORBA Data Module from the Multitier page of the new items dialog. You will see the CORBA Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TCorbaDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TCorbaDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note You can add your own properties and methods to your new interface. For more information on adding to your data module's interface, see "Extending the application server's interface" on page 25-16.

The CORBA Data Module wizard lets you specify how you want your server application to create instances of the remote data module. You can choose either shared or instance-per-client.

- When you choose shared, your application creates a single instance of the remote data module that handles all client requests. This is the model used in traditional CORBA development.
- When you choose instance-per-client, a new remote data module instance is created for each client connection. This instance persists until its timeout period elapses with no messages from the client. This allows the server to free instances when they are no longer used by clients, but holds the risk that the server may be freed prematurely if the client does not use the server's interface for a long time.

Note Unlike instancing for COM servers, where the model determines the number of instances of the process that run, with CORBA, instancing determines the number of instances created of your object. They are all created within a single instance of the server executable.

In addition to the instancing model, you must specify the threading model in the CORBA Data Module wizard. You can choose Single- or Multi-threaded.

- If you choose Single-threaded, each remote data module instance is guaranteed to receive only one client request at a time. You can safely access the objects contained in your remote data module. However, you must guard against thread conflicts when you use global variables or objects not contained in the remote data module.
- If you choose Multi-threaded, each client connection has its own dedicated thread. However, your application may be called by multiple clients simultaneously, each on a separate thread. You must guard against simultaneous access of instance data as well as global memory. Writing Multi-threaded servers is tricky when you are using a shared remote data module instance, because you must protect all use of objects contained in your remote data module.

Extending the application server's interface

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is created for you automatically by the wizard when you create the remote data module.

To add to the remote data module's interface, you can

- Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you are adding a procedure, function, or property, and enter its syntax. When you click OK, you will be positioned in the code editor on the implementation of your new interface member.
- Use the type library editor. Select the interface for your application server in the type library editor, and click the tool button for the type of interface member (method or property) that you are adding. Give your interface member a name in the Attributes page, specify parameters and type in the Parameters page, and then refresh the type library. See Chapter 34, "Working with type libraries" for more information about using the type library editor, Note that many of the features you can specify in the type library editor (such as help context, version, and so on) do not apply to CORBA interfaces. Any values you specify for these in the type library editor are ignored.

Note Neither of these approaches works if you are implementing *TSoapDataModule*. For *TSoapDataModule* descendants, you must edit the server interface directly.

What Delphi does when you add new entries to the interface using the type library editor or the Add To Interface command depends on whether you are creating a COM-based (*TRemoteDataModule* or *TMTSDataModule*) or CORBA (*TCorbaDataModule*) server.

- When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).
- When you add to a CORBA interface, your changes are reflected in your unit source code and the automatically generated _TLB unit. The _TLB unit is added to the **uses** clause of your unit. You must add this unit to the **uses** clause in your client application if you want to take advantage of early binding. In addition, you can save an .IDL file from the type library editor using the Export to IDL button. The .IDL file is needed for registering the interface with the Interface Repository and Object Activation Daemon.

Note You must explicitly save the TLB file by choosing Refresh in the type library editor and then saving the changes from the IDE.

Once you have added to your remote data module's interface, locate the properties and methods that were added to your remote data module's implementation. Add code to finish this implementation by filling in the bodies of the new methods.

Client applications call your interface extensions using the *AppServer* property of their connection component. For more information on how to do this, see "Calling server interfaces" on page 25-29.

Adding callbacks to the application server's interface

You can allow the application server to call your client application by introducing a callback. To do this, the client application passes an interface to one of the application server's methods, and the application server later calls this method as needed.

However, if your extensions to the remote data module's interface include callbacks, you can't use an HTTP or SOAP-based connection. *TWebConnection* does not support callbacks. If you are using a socket-based connection, client applications must indicate whether they are using callbacks by setting the *SupportCallbacks* property. All other types of connection automatically support callbacks.

Extending a transactional application server's interface

When using transactions or just-in-time activation, you must be sure all new methods call *SetComplete* to indicate when they are finished. This allows transactions to complete and permits the remote data module to be deactivated.

Furthermore, you can't return any values from your new methods that allow the client to communicate directly with objects or interfaces on the application server unless they provide a safe reference. If you are using a stateless MTS data module, neglecting to use a safe reference can lead to crashes because you can't guarantee that the remote data module is active. For more information on safe references, see "Passing object references" on page 39-20.

Managing transactions in multi-tiered applications

When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back.

In addition, you can add transaction support to your server application by adding a database connection component or managing the transaction directly by sending SQL to the database server. This works the same way that you would manage transactions in a two-tiered application. For more information about this sort of transaction control, see "Managing transactions" on page 17-5.

If you have a transactional data module, you can broaden your transaction support by using MTS or COM+ transactions. These transactions can include any of the business logic on your application server, not just the database access. In addition, because they support two-phase commits, they can span multiple databases.

Only the BDE- and ADO-based data access components support two-phase commit. Do not use InterbaseExpress or dbExpress components if you want to have transactions that span multiple databases.

Warning When using the BDE, two-phase commit is fully implemented only on Oracle7 and MS-SQL databases. If your transaction involves multiple databases, and some of them are remote servers other than Oracle7 or MS-SQL, your transaction runs a small risk of only partially succeeding. Within any one database, however, you will always have transaction support.

By default, all *IAppServer* calls on a transactional data module are transactional. You need only set the transaction attribute of your data module to indicate that it must participate in transactions. In addition, you can extend the application server's interface to include method calls that encapsulate transactions that you define.

If your transaction attribute indicates that the remote data module requires a transaction, then every time a client calls a method on its interface, it is automatically wrapped in a transaction. All client calls to your application server are then enlisted in that transaction until you indicate that the transaction is complete. These calls either succeed as a whole or are rolled back.

Note Do not combine MTS or COM+ transactions with explicit transactions created by a database connection component or using explicit SQL commands. When your transactional data module is enlisted in a transaction, it automatically enlists all of your database calls in the transaction as well.

For more information about using MTS and COM+ transactions, see "MTS and COM+ transaction support" on page 39-8.

Supporting master/detail relationships

You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see "Creating master/detail relationships" on page 18-34.

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see "Limiting records with parameters" on page 23-28.)
- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database connection component to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this when providing from datasets, set up a master/detail relationship between the datasets on the application server. Then set the *DataSet* property of your provider component to the master table. To use nested tables to represent master/detail relationships when providing from XML documents, use a transformation file that defines the nested detail sets.

When clients call the *GetRecords* method of the provider, it automatically includes the detail dataset as a DataSet field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

Supporting state information in remote data modules

The *IAppServer* interface, which controls all communication between client datasets and providers on the application server, is mostly stateless. When an application is stateless, it does not "remember" anything that happened in previous calls by the client. This stateless quality is useful if you are pooling database connections in a transactional data module, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with just-in-time activation, object pooling, or typical CORBA servers. SOAP data modules must be stateless.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching, the provider on the application server must "remember" information from previous calls (the current record).

Before and after any calls to the *IAppServer* interface that the client dataset makes (AS_*ApplyUpdates*, AS_*Execute*, AS_*GetParams*, AS_*GetRecords*, or AS_*RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless.

For example, consider a dataset that represents the following parameterized query:

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

To enable incremental fetching in a stateless application server, you can do the following:

- When the provider packages a set of records in a data packet, it notes the value of CUST_NO on the last record in the packet:

```
TRemoteDataModule1.DataSetProvider1GetData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
  DataSet.Last; { move to the last record }
  with Sender as TDataSetProvider do
    Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
end;
```

- The provider sends this last CUST_NO value to the client after sending the data packet:

```
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender: TObject;
                   var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {send the last value of CUST_NO }
end;
```

- On the client, the client dataset saves this last value of CUST_NO:

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {save the last value of CUST_NO }
end;
```

- Before fetching a data packet, the client sends the last value of CUST_NO it received:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
  begin
    if not Active then Exit;
    OwnerData := Tag; { Send last value of CUST_NO to application server }
  end;
end;
```

- Finally, on the server, the provider uses the last CUST_NO sent as a minimum value in the query:

```
TRemoteDataModule1.DataSetProvider1BeforeGetRecords(Sender: TObject;
                    var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
      begin
        Params.ParamValues['MinVal'] := OwnerData;
        Refresh; { force the query to reexecute }
    end;
end;
```

Using multiple remote data modules

You may want to structure your application server so that it uses multiple remote data modules. Using multiple remote data modules lets you partition your code, organizing a large application server into multiple self-contained units, where each unit is relatively self-contained.

Although you can always create multiple remote data modules on the application server that function independently, Delphi provides support for a model where you have one main “parent” remote data module that dispatches connections from clients to other “child” remote data modules.

To create the parent remote data module, you must extend its *IAppServer* interface, adding properties that expose the interfaces of the child remote data modules. That is, for each child remote data module, add a property to the parent data module’s interface whose value is the *IAppServer* interface for the child data module. The property getter should look something like the following:

```
function ParentRDM.Get_ChildRDM: IChildRDM;
begin
  {note the parent RDM uses a factory component defined in the child RDM’s unit.
  This is more efficient if it must create several children for different clients }
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.ParentRDM := Self;
end;
```

For information about extending the parent remote data module's interface, see "Extending the application server's interface" on page 25-16.

Tip You may also want to extend the interface for each child data module, exposing the parent data module's interface, or the interfaces of the other child data modules. This lets the various data modules in your application server communicate more freely with each other.

Once you have added properties that represent the child remote data modules to the main remote data module, client applications do not need to form separate connections to each remote data module on the application server. Instead, they share a single connection to the parent remote data module, which then dispatches messages to the "child" data modules. Because each client application uses the same connection for every remote data module, the remote data modules can share a single database connection, conserving resources. For information on how child applications share a single connection, see "Connecting to an application server that uses multiple data modules" on page 25-30.

Registering the application server

Before client applications can locate and use an application server, it must be registered or installed. (This is not strictly true for CORBA application servers, although registration is still recommended.)

- If the application server uses DCOM, HTTP, or sockets as a communication protocol, it acts as an Automation server and must be registered like any other COM server. For information about registering a COM server, see "Registering a COM object" on page 36-16.
- If you are using a transactional data module, you do not register the application server. Instead, you install it with MTS or COM+. For information about installing transactional objects, see "Installing transactional objects" on page 39-22.
- When the application server uses SOAP, the application must be a Web Service application. As such, it must be registered with your Web Server, so that it receives incoming HTTP messages. In addition, if you want clients that are not written using Delphi to access any of the interfaces in your application, you can publish a WSDL document that describes the invokable interfaces in your application. For information about exporting a WSDL document for a Web Service application, see "Generating WSDL documents for a Web Service application" on page 31-7.
- When the application server uses CORBA, registration is optional. If you want to allow client applications to use dynamic binding to your interface, you must install the server's interface in the Interface Repository. In addition, if you want to allow client applications to launch the application server when it is not already running, it must be registered with the OAD (Object Activation Daemon).

Creating the client application

In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.

To create a multi-tiered client application, start a new project and follow these steps:

1. Add a new data module to the project.
2. Place a connection component on the data module. The type of connection component you add depends on the communication protocol you want to use. See "The structure of the client application" on page 25-4 for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see "Connecting to the application server" on page 25-23.
4. Set the other connection component properties as needed for your application. For example, you might set the *ObjectBroker* property to allow the connection component to choose dynamically from several servers. For more information about using the connection components, see "Managing server connections" on page 25-28
5. Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see Chapter 23, "Using client datasets."
6. Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property's drop-down list.
7. Continue in the same way you would create any other database application. There are a few additional features available to clients of multi-tiered applications:
   - Your application may want to make direct calls to the application server. "Calling server interfaces" on page 25-29 describes how to do this.
   - You may want to use the special features of client datasets that support their interaction with the provider components. These are described in "Using a client dataset with a provider" on page 23-23.

Connecting to the application server

To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the DataSnap page of the Component palette.

Use a connection component to

- Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See “Choosing a connection protocol” on page 25-8 for details on the benefits and limitations of the available protocols.
- Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol.
- Identify the application server on the server machine.

  If you are not using CORBA, identify the server using the *ServerName* or *ServerGUID* property. *ServerName* identifies the base name of the class you specify when creating the remote data module on the application server. See “Setting up the remote data module” on page 25-13 for details on how this value is specified on the server. If the server is registered or installed on the client machine, or if the connection component is connected to the server machine, you can set the *ServerName* property at design time by choosing from a drop-down list in the Object Inspector. *ServerGUID* specifies the GUID of the remote data module’s interface. You can look up this value using the type library editor.

  If you are using CORBA, identify the server using the *RepositoryID* property. *RepositoryID* specifies the Repository ID of the application server’s factory interface, which appears as the third argument in the call to *TCorbaVCLComponentFactory.Create* that is automatically added to the initialization section of the CORBA server’s implementation unit. You can also set this property to the base name of the CORBA data module’s interface (the same string as the *ServerName* property for other connection components), and it is automatically converted into the appropriate Repository ID for you.
- Manage server connections. Connection components can be used to create or drop connections and to call application server interfaces.

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the connection component to identify the application server by name, specify a server machine, and use the application server interface.

Specifying a connection using DCOM

When using DCOM to communicate with the application server, client applications include a *TDCOMConnection* component for connecting to the application server. *TDCOMConnection* uses the *ComputerName* property to identify the machine on which the server resides.

When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*.

Note Even when there is a system registry entry for the application server, you can specify *ComputerName* to override this entry. This can be especially useful during development, testing, and debugging.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *ComputerName*. For more information, see "Brokering connections" on page 25-27.

If you supply the name of a host computer or server that cannot be found, the DCOM connection component raises an exception when you try to open the connection.

Specifying a connection using sockets

You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a *TSocketConnection* component for connecting to the application server.

TSocketConnection identifies the server machine using the IP Address or host name of the server system, and the port number of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see "Describing sockets" on page 32-3.

Three properties of *TSocketConnection* specify this information:

- *Address* specifies the IP Address of the server.
- *Host* specifies the host name of the server.
- *Port* specifies the port number of the socket dispatcher program on the application server.

Address and *Host* are mutually exclusive. Setting one unsets the value of the other. For information on which one to use, see "Describing the host" on page 32-4.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *Address* or *Host*. For more information, see "Brokering connections" on page 25-27.

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher programs supplied with Delphi. If the socket dispatcher has been configured to use a different port, set the *Port* property to match that value.

Note You can configure the port of the socket dispatcher while it is running by right-clicking the Borland Socket Server tray icon and choosing Properties.

Although socket connections do not provide for using security protocols, you can customize the socket connection to add your own encryption. To do this

1. Create a COM object that supports the *IDataIntercept* interface. This is an interface for encrypting and decrypting data.
2. Use *TPacketInterceptFactory* as the class factory for this object. If you are using a wizard to create the COM object in step 1, replace the line in the initialization section that says `TComponentFactory.Create(...)` with `TPacketInterceptFactory.Create(...)`.
3. Register your new COM server on the client machine.

4 Set the *InterceptName* or *InterceptGUID* property of the socket connection component to specify this COM object. If you used *TPacketInterceptFactory* in step 2, your COM server appears in the drop-down list of the Object Inspector for the *InterceptName* property.

5 Finally, right click the Borland Socket Server tray icon, choose Properties, and on the properties tab set the Intercept Name or Intercept GUID to the ProgId or GUID for the interceptor.

This mechanism can also be used for data compression and decompression.

Specifying a connection using HTTP

You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a *TWebConnection* component for connecting to the application server.

The Web connection component establishes a connection to the Web server application (httpsrvr.dll), which in turn communicates with the application server. *TWebConnection* locates httpsrvr.dll using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server and httpsrvr.dll, and the path to the Web server application (httpsrvr.dll). Specify this value using the *URL* property.

Note When using *TWebConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the *UserName* and *Password* properties so that the connection component can log on.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *URL*. For more information, see “Brokering connections” on page 25-27.

Specifying a connection using SOAP

You can establish a connection to a SOAP application server using the *TSoapConnection* component. *TSoapConnection* is very similar to *TWebConnection*, because it also uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security to communicate with a server that is protected by a firewall.

The SOAP connection component establishes a connection to a Web server application that implements the *IAppServer* interface as a Web Service. *TSoapConnection* locates this Web Server application using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server, the name of the Web Service application, and a path that matches the path name of the *THTTPSoapDispatcher* on the application server. Specify this value using the *URL* property.

Note When using *TSoapConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the *UserName* and *Password* properties so that the connection component can log on.

Specifying a connection using CORBA

Only the *RepositoryID* property is necessary in order to specify a CORBA connection. This is because a Smart Agent on the local network automatically locates an available server for your CORBA client.

However, you can limit the possible servers to which your client application connects by the other properties of the CORBA connection component. If you want to specify a particular server machine, rather than letting the CORBA Smart Agent locate any available server, use the *HostName* property. If there is more than one object instance that implements your server interface, you can specify which object you want to use by setting the *ObjectName* property.

The *TCorbaConnection* component obtains an interface to the CORBA data module on the application server in one of two ways:

- If you are using early (static) binding, you must add the _TLB.pas file (generated by the type library editor) to your client application. Early binding is highly recommended, both for compile-time type checking and because it is much faster than late (dynamic) binding.
- If you are using late (dynamic) binding, the interface must be registered with the Interface Repository.

For more information on early vs. late binding, see “Calling server interfaces” on page 25-29.

Brokering connections

If you have multiple servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so on, until a connection is formed.

Once the connection component has formed a connection with a name supplied by the broker, it saves that name as the value of the appropriate property (*ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL*). If the connection component closes the connection later, and then needs to reopen the connection, it tries using this property value, and only requests a new name from the broker if the connection fails.

Use an Object Broker by specifying the *ObjectBroker* property of your connection component. When the *ObjectBroker* property is set, the connection component does not save the value of *ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL* to disk.

Note Do not use the *ObjectBroker* property with CORBA connections. CORBA has its own brokering mechanism.

Managing server connections

The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface.

Connecting to the server

To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in "Connecting to the application server" on page 25-23. Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the connection component, you can open the connection by setting the *Connected* property of the connection component to *True*.

Before a connection component establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

Dropping or changing a server connection

A connection component drops a connection to the application server when you

- set the *Connected* property to *False*.
- free the connection component. A connection object is automatically freed when a user closes the client application.
- change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

Note Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

Calling server interfaces

Applications do not need to call the *IAppServer* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface, you need a way to call those extensions using the connection created by your connection component. Unless you are using SOAP, you can do this using the *AppServer* property of the connection component. For information about extending the application server's interface, see "Extending the application server's interface" on page 25-16.

AppServer is a Variant that represents the application server's interface. You can call an interface method using *AppServer* by writing a statement such as

```
MyConnection.AppServer.SpecialMethod(x,y);
```

However, this technique provides late (dynamic) binding of the interface call. That is, the *SpecialMethod* procedure call is not bound until runtime when the call is executed. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the server to set up interface calls before they are invoked.

When you are using DCOM or CORBA as a communications protocol, you can use early binding of *AppServer* calls. Use the **as** operator to cast the *AppServer* variable to the *IAppServer* descendant you created when you created the remote data module. For example:

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

To use early binding under DCOM, the server's type library must be registered on the client machine. You can use TRegsvr.exe, which ships with Delphi to register the type library.

Note See the TRegSvr demo (which provides the source for TRegsvr.exe) for an example of how to register the type library programmatically.

To use early binding with CORBA, you must add the _TLB unit that is generated by the type library editor to your project. To do this, add this unit to the **uses** clause of your unit.

When you are using TCP/IP or HTTP, you can't use true early binding, but because the remote data module uses a dual interface, you can use the application server's dispinterface to improve performance over simple late-binding. The dispinterface has the same name as the remote data module's interface, with the string 'Disp' appended. You can assign the *AppServer* property to a variable of this type to obtain the dispinterface. Thus:

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
...
  TempInterface.SpecialMethod(x,y);
...
end;
```

Note To use the dispinterface, you must add the _TLB unit that is generated when you save the type library to the **uses** clause of your client module.

If you are using SOAP, you can't use the *AppServer* property. Instead, you must use a remote interfaced object (*THTTPRio*) and make early-bound calls. As with all early-bound calls, the client application must know the application server's interface declaration at compile time. You can add this to your client application either by adding the same unit the server uses to declare and register the interface to the client's uses clause, or you can reference a WSDL document that describes the interface. For information on importing a WSDL document that describes the server interface, see "Importing WSDL documents" on page 31-8.

Note The unit that declares the server interface must also register it with the invocation registry. For details on how to register invokable interfaces, see "Defining invokable interfaces" on page 31-3.

Once your application uses the server unit that declares and registers the interface, or you have imported a WSDL document to generate such a unit, create an instance of *THTTPRio* for the desired interface:

```
X := THTTPRio.Create(nil);
```

Next, assign the URL that your connection component uses to the remote interfaced object:

```
X.URL := SoapConnection1.URL;
```

You can then use the **as** operator to cast the instance of *THTTPRio* to the application server's interface:

```
InterfaceVariable := X as IMyAppServer;
InterfaceVariable.SpecialMethod(x,y);
```

Connecting to an application server that uses multiple data modules

If the application server uses a main "parent" remote data module and several child remote data modules, as described in "Using multiple remote data modules" on page 25-21, then you need a separate connection component for every remote data module on the application server. Each connection component represents the connection to a single remote data module.

While it is possible to have your client application form independent connections to each remote data module on the application server, it is more efficient to use a single connection to the application server that is shared by all the connection components. That is, you add a single connection component that connects to the "main" remote data module on the application server, and then, for each "child" remote data module, add an additional component that shares the connection to the main remote data module.

1 For the connection to the main remote data module, add and set up a connection component as described in "Connecting to the application server" on page 25-23. The only limitation is that you can't use a CORBA or SOAP connection.

2 For each child remote data module, use a *TSharedConnection* component.

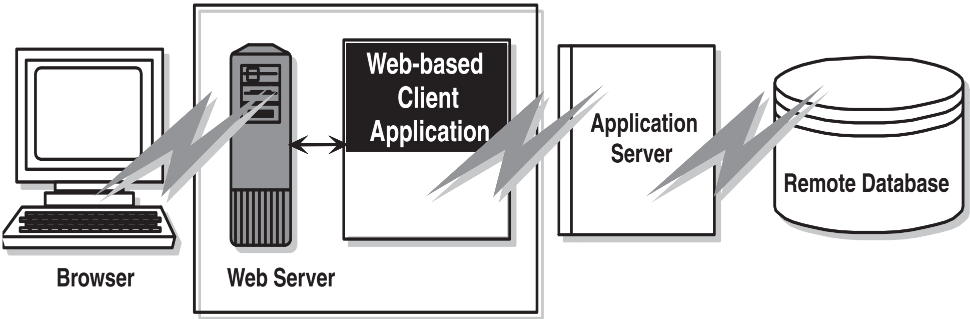
  - Set its *ParentConnection* property to the connection component you added in step 1. The *TSharedConnection* component shares the connection that this main connection establishes.
  - Set its *ChildName* property to the name of the property on the main remote data module's interface that exposes the interface of the desired child remote data module.

When you assign the *TSharedConnection* component placed in step 2 as the value of a client dataset's *RemoteServer* property, it works as if you were using an entirely independent connection to the child remote data module. However, the *TSharedConnection* component uses the connection established by the component you placed in step 1.

Writing Web-based client applications

If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web application that acts simultaneously as a client to an application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in Figure 25.1.

Figure 25.1 Web-based multi-tiered database application



There are two approaches that you can take to build the Web application:

- You can combine the multi-tiered database architecture with Delphi's ActiveX support to distribute the client application as an ActiveX control. This allows any browser that supports ActiveX to run your client application as an in-process server.

- You can use XML data packets to build an InternetExpress application. This allows browsers that supports javascript to interact with your client application through html pages.

These two approaches are very different. Which one you choose depends on the following considerations:

- Each approach relies on a different technology (ActiveX vs. javascript and XML). Consider what systems your end users will use. The first approach requires a browser to support ActiveX (which limits clients to a Windows platform). The second approach requires a browser to support javascript and the DHTML capabilities introduced by Netscape 4 and Internet Explorer 4.
- ActiveX controls must be downloaded to the browser to act as an in-process server. As a result, the clients using an ActiveX approach require much more memory than the clients of an HTML-based application.
- The InternetExpress approach can be integrated with other HTML pages. An ActiveX client must run in a separate window.
- The InternetExpress approach uses standard HTTP, thereby avoiding any firewall issues that confront an ActiveX application.
- The ActiveX approach provides greater flexibility in how you program your application. You are not limited by the capabilities of the javascript libraries. The client datasets used in the ActiveX approach surface more features (such as filters, ranges, aggregation, optional parameters, delayed fetching of BLOBs or nested details, and so on) than the XML brokers used in the InternetExpress approach.

Caution Your Web client application may look and act differently when viewed from different browsers. Test your application with the browsers you expect your end-users to use.

Distributing a client application as an ActiveX control

The multi-tiered database architecture can be combined with Delphi's ActiveX features to distribute a client application as an ActiveX control.

When you distribute your client application as an ActiveX control, create the application server as you would for any other multi-tiered application. The only limitation is that you will want to use DCOM, HTTP, SOAP, or sockets as a communications protocol, because you can't count on client machines having installed the CORBA runtime software. For details on creating the application server, see "Creating the application server" on page 25-11.

When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See "Creating an Active Form for the client application" for details.

Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your client application, the Web server must be running on the machine that has the client application.

If the client application uses DCOM to communicate between the client application and the application server, the machine with the Web browser must be enabled to work with DCOM. If the machine with the Web browser is a Windows 95 machine, it must have installed DCOM95, which is available from Microsoft.

Creating an Active Form for the client application

1. Because the client application will be deployed as an ActiveX control, you must have a Web server that runs on the same system as the client application. You can use a ready-made server such as Microsoft's Personal Web server or you can write your own using the socket components described in Chapter 32, "Working with sockets."
2. Create the client application following the steps described in "Creating the client application" on page 25-23, except start by choosing File | New | Active Form, rather than beginning the client project as an ordinary Delphi project.
3. If your client application uses a data module, add a call to explicitly create the data module in the active form initialization.
4. When your client application is finished, compile the project, and select Project | Web Deployment Options. In the Web Deployment Options dialog, you must do the following:
   1. On the Project page, specify the Target directory, the URL for the target directory, and the HTML directory. Typically, the Target directory and the HTML directory will be the same as the projects directory for your Web Server. The target URL is typically the name of the server machine that is specified in the Windows Network | DNS settings.
   2. On the Additional Files page, include midas.dll with your client application.
5. Finally, select Project | WebDeploy to deploy the client application as an active form.

Any Web browser that can run Active forms can run your client application by specifying the .HTM file that was created when you deployed the client application. This .HTM file has the same name as your client application project, and appears in the directory specified as the Target directory.

Building Web applications using InternetExpress

A client application can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries of database functions, and Delphi's Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. These applications make up Delphi's InternetExpress support.

Before building an InternetExpress application, you should understand Delphi's Web server application architecture. This is described in Chapter 27, "Creating Internet applications."

An InternetExpress application extends the basic Web server application architecture to act as the client of an application server. InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML governs the layout and appearance of the pages seen by end users in their browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in these XML data packets on the client machine.

If the InternetExpress application uses DCOM to connect to the application server, you must take additional steps to ensure that the application server grants access and launch permissions to its clients. See "Granting permission to access and launch the application server" on page 25-36 for details.

Tip You can create an InternetExpress application to provide Web browsers with "live" data even if you do not have an application server. Simply add the provider and its dataset to the Web module.

Building an InternetExpress application

The following steps describe one way to build a Web application using InternetExpress. The result is an application that creates HTML pages that let users interact with the data from an application server via a javascript-enabled Web browser. You can also build an InternetExpress application using the Site Express architecture by using the InternetExpress page producer (*TInetXPageProducer*).

1. Choose File | New to display the New Items dialog box, and on the New page select Web Server application. This process is described in "Creating Web server applications with Web Broker" on page 28-1.
2. From the DataSnap page of the component palette, add a connection component to the Web Module that appears when you create a new Web server application. The type of connection component you add depends on the communication protocol you want to use. See "Choosing a connection protocol" on page 25-8 for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see "Connecting to the application server" on page 25-23.
4. Instead of a client dataset, add an XML broker from the InternetExpress page of the component palette to the Web module. Like *TClientDataSet*, *TXMLBroker* represents the data from a provider on the application server and interacts with the application server through its *IAppServer* interface. However, unlike client datasets, XML brokers request data packets as XML instead of as OleVariants and interact with InternetExpress components instead of data controls.
5. Set the *RemoteServer* property of the XML broker to point to the connection component you added in step 2. Set the *ProviderName* property to indicate the provider on the application server that provides data and applies updates. For more information about setting up the XML broker, see "Using an XML broker" on page 25-36.

6 Add an InternetExpress page producer (*TInetXPageProducer*) to the Web module for each separate page that users will see in their browsers. For each page producer, you must set the *IncludePathURL* property to indicate where it can find the javascript libraries that augment its generated HTML controls with data management capabilities.

7 Right-click a Web page and choose Action Editor to display the Action editor. Add action items for every message you want to handle from browsers. Associate the page producers you added in step 6 with these actions by setting their *Producer* property or writing code in an *OnAction* event handler. For more information on adding action items using the Action editor, see “Adding actions to the dispatcher” on page 28-4.

8 Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipsis button in the Object Inspector next to the *WebPageItems* property.) In this editor you can add Web Items to design the pages that users see in their browsers. For more information about designing Web pages for your InternetExpress application, see “Creating Web pages with an InternetExpress page producer” on page 25-38.

9 Build your Web application. Once you install this application with your Web server, browsers can call it by specifying the name of the application as the scriptname portion of the URL and the name of the Web Page component as the pathinfo portion.

Using the javascript libraries

The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship with Delphi:

Table 25.3 Javascript libraries

Library	Description
xmldom.js	This library is a DOM-compatible XML parser written in javascript. It allows parsers that do not support XML to use XML data packets. Note that this does not include support for XML Islands, which are supported by IE5 and later.
xmldb.js	This library defines data access classes that manage XML data packets and XML delta packets.
xmldisp.js	This library defines classes that associate the data access classes in xmldb with HTML controls in the HTML page.
xmlerrdisp.js	This library defines classes that can be used when reconciling update errors. These classes are not used by any of the built-in InternetExpress components, but are useful when writing a Reconcile producer.
xmlshow.js	This library includes functions to display formatted XML data packets and XML delta packets. This library is not used by any of the InternetExpress components, but is useful when debugging.

These libraries can be found in the Source/Webmidas directory. Once you have installed these libraries, you must set the *IncludePathURL* property of all InternetExpress page producers to indicate where they can be found.

It is possible to write your own HTML pages using the javascript classes provided in these libraries instead of using Web items to generate your Web pages. However, you must ensure that your code does not do anything illegal, as these classes include minimal error checking (so as to minimize the size of the generated Web pages).

The classes in the javascript libraries are an evolving standard, and are updated regularly. If you want to use them directly rather than relying on Web items to generate the javascript code, you can get the latest versions and documentation of how to use them from CodeCentral available through community.borland.com.

Granting permission to access and launch the application server

Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with `EOLE_ACCESS_ERROR`.

Note Because the application server runs under this guest account, it can't be shut down by other accounts.

To grant the Web application access and launch permissions, run DCOMCnfg.exe, which is located in the System32 directory of the machine that runs the application server. The following steps describe how to configure your application server:

1. When you run DCOMCnfg, select your application server in the list of applications on the Applications page.
2. Click the Properties button. When the dialog changes, select the Security page.
3. Select Use Custom Access Permissions, and press the Edit button. Add the name IUSR_computername to the list of accounts with access permission, where computername is the name of the machine that runs the Web application.
4. Select Use Custom Launch Permissions, and press the Edit button. Add IUSR_computername to this list as well.
5. Click the Apply button.

Using an XML broker

The XML broker serves two major functions:

- It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application.
- It receives updates in the form of XML delta packets from browsers and applies them to the application server.

Fetching XML data packets

Before the XML broker can supply XML data packets to the components that generate HTML pages, it must fetch them from the application server. To do this, it uses the *IAppServer* interface of the application server, which it acquires through a connection component. You must set the following properties so that the XML producer can use the application server's *IAppServer* interface:

- Set the *RemoteServer* property to the connection component that establishes the connection to the application server and gets its *IAppServer* interface. At design time, you can select this value from a drop-down list in the object inspector.
- Set the *ProviderName* property to the name of the provider component on the application server that represents the dataset for which you want XML data packets. This provider both supplies XML data packets and applies updates from XML delta packets. At design time, if the *RemoteServer* property is set and the connection component has an active connection, the Object Inspector displays a list of available providers. (If you are using a DCOM connection the application server must also be registered on the client machine).

Two properties let you indicate what you want to include in data packets:

- You can limit the number of records that are added to the data packet by setting the *MaxRecords* property. This is especially important for large datasets because InternetExpress applications send the entire data packet to client Web browsers. If the data packet is too large, the download time can become prohibitively long.
- If the provider on the application server represents a query or stored procedure, you may want to provide parameter values before obtaining an XML data packet. You can supply these parameter values using the *Params* property.

The components that generate HTML and javascript for the InternetExpress application automatically use the XML broker's XML data packet once you set their *XMLBroker* property. To obtain the XML data packet directly in code, use the *RequestRecords* method.

Note When the XML broker supplies a data packet to another component (or when you call *RequestRecords*), it receives an *OnRequestRecords* event. You can use this event to supply your own XML string instead of the data packet from the application server. For example, you could fetch the XML data packet from the application server using *GetXMLRecords* and then edit it before supplying it to the emerging Web page.

Applying updates from XML delta packets

When you add the XML broker to the Web module (or data module containing a *TWebDispatcher*), it automatically registers itself with the Web dispatcher as an auto-dispatching object. This means that, unlike other components, you do not need to create an action item for the XML broker in order for it to respond to update messages from a Web browser. These messages contain XML delta packets that should be applied to the application server. Typically, they originate from a button that you create on one of the HTML pages produced by the Web client application.

So that the dispatcher can recognize messages for the XML broker, you must describe them using the *WebDispatch* property. Set the *PathInfo* property to the path portion of the URL to which messages for the XML broker are sent. Set *MethodType* to the value of the method header of update messages addressed to that URL (typically *mtPost*). If you want to respond to all messages with the specified path, set *MethodType* to *mtAny*. If you don't want the XML broker to respond directly to update messages (for example, if you want to handle them explicitly using an action item), set the *Enabled* property to *False*. For more information on how the Web dispatcher determines which component handles messages from the Web browser, see “Dispatching request messages” on page 28-5.

When the dispatcher passes an update message on to the XML broker, it passes the updates on to the application server and, if there are update errors, receives an XML delta packet describing all update errors. Finally, it sends a response message back to the browser, which either redirects the browser to the same page that generated the XML delta packet or sends it some new content.

A number of events allow you to insert custom processing at all steps of this update process:

1. When the dispatcher first passes the update message to the XML broker, it receives a *BeforeDispatch* event, where you can preprocess the request or even handle it entirely. This event allows the XML broker to handle messages other than update messages.
2. If the *BeforeDispatch* event handler does not handle the message, the XML broker receives an *OnRequestUpdate* event, where you can apply the updates yourself rather than using the default processing.
3. If the *OnRequestUpdate* event handler does not handle the request, the XML broker applies the updates and receives a delta packet containing any update errors.
4. If there are no update errors, the XML broker receives an *OnGetResponse* event, where you can create a response message that indicates the updates were successfully applied or sends refreshed data to the browser. If the *OnGetResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker sends a response that redirects the browser back to the document that generated the delta packet.
5. If there are update errors, the XML broker receives an *OnGetErrorResponse* event instead. You can use this event to try to resolve update errors or to generate a Web page that describes them to the end user. If the *OnGetErrorResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker calls on a special content producer called the *ReconcileProducer* to generate the content of the response message.
6. Finally, the XML broker receives an *AfterDispatch* event, where you can perform any final actions before sending a response back to the Web browser.

Creating Web pages with an InternetExpress page producer

Each InternetExpress page producer generates an HTML document that appears in the browsers of your application’s clients. If your application includes several separate Web documents, use a separate page producer for each of them.

The InternetExpress page producer (*TInetXPageProducer*) is a special page producer component. As with other page producers, you can assign it as the *Producer* property of an action item or call it explicitly from an *OnAction* event handler. For more information about using content producers with action items, see “Responding to request messages with action items" on page 28-7. For more information about page producers, see "Using page producer components" on page 28-13.

Unlike most page producers, the InternetExpress page producer has a default template as the value of its *HTMLDoc* property. This template contains a set of HTML-transparent tags that the InternetExpress page producer uses to assemble an HTML document (with embedded javascript and XML) including content produced by other components. Before it can translate all of the HTML-transparent tags and assemble this document, you must indicate the location of the javascript libraries used for the embedded javascript on the page. This location is specified by setting the *IncludePathURL* property.

You can specify the components that generate parts of the Web page using the Web page editor. Display the Web page editor by double-clicking the Web page component or clicking the ellipsis button next to the *WebPageItems* property in the Object Inspector.

The components you add in the Web page editor generate the HTML that replaces one of the HTML-transparent tags in the InternetExpress page producer's default template. These components become the value of the *WebPageItems* property. After adding the components in the order you want them, you can customize the template to add your own HTML or change the default tags.

Using the Web page editor

The Web page editor lets you add Web items to your InternetExpress page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a InternetExpress page producer component.

Note You must have Internet Explorer 4 or better installed to use the Web page editor.

The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of the Web items, indicating how they are nested. On the right, you can see the Web items included by the currently selected item. When you select a component in the top of the Web page editor, you can set its properties using the Object Inspector.

Click the New Item button to add a subitem to the currently selected item. The Add Web Component dialog lists only those items that can be added to the currently selected item.

The InternetExpress page producer can contain one of two types of item, each of which generates an HTML form:

- *TDataForm*, which generates an HTML form for displaying data and the controls that manipulate that data or submit updates.

  Items you add to *TDataForm* display data in a multi-record grid (*TDataGrid*) or in a set of controls each of which represents a single field from a single record (*TFieldGroup*). In addition, you can add a set of buttons to navigate through data or post updates (*TDataNavigator*), or a button to apply updates back to the Web client (*TApplyUpdatesButton*). Each of these items contains subitems to represent individual fields or buttons. Finally, as with most Web items, you can add a layout grid (*TLayoutGroup*), that lets you customize the layout of any items it contains.

- *TQueryForm*, which generates an HTML form for displaying or reading application-defined values. For example, you can use this form for displaying and submitting parameter values.

  Items you add to *TQueryForm* display application-defined values(*TQueryFieldGroup*) or a set of buttons to submit or reset those values (*TQueryButtons*). Each of these items contains subitems to represent individual values or buttons. You can also add a layout grid to a query form, just as you can to a data form.

The bottom of the Web page editor displays the generated HTML code and lets you see what it looks like in a browser (IE4).

Setting Web item properties

The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML.

When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the *XMLBroker* property associates the Web item with the XML broker that manages the data packet. You can further specify a dataset that is contained in a dataset field of that data packet using the *XMLDataSetField* property. If the Web item represents a specific field or parameter value, the Web item has a *FieldName* or *ParamName* property.

You can apply a style attribute to any Web item, thereby influencing the overall appearance of all the HTML it generates. Styles and style sheets are part of the HTML 4 standard. They allow an HTML document to define a set of display attributes that apply to a tag and everything in its scope. Web items offer a flexible selection of ways to use them:

- The simplest way to use styles is to define a style attribute directly on the Web item. To do this, use the *Style* property. The value of *Style* is simply the attribute definition portion of a standard HTML style definition, such as
  `color: red.`
- You can also define a style sheet that defines a set of style definitions. Each definition includes a style selector (the name of a tag to which the style always applies or a user-defined style name) and the attribute definition in curly braces:

  ```
  H2 B  {color: red}
  .MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
  ```

  The entire set of definitions is maintained by the InternetExpress page producer as its *Styles* property. Each Web item can then reference the styles with user-defined names by setting its *StyleRule* property.

- If you are sharing a style sheet with other applications, you can supply the style definitions as the value of the InternetExpress page producer's *StylesFile* property instead of the *Styles* property. Individual Web items still reference styles using the *StyleRule* property.

Another common property of Web items is the *Custom* property. *Custom* includes a set of options that you add to the generated HTML tag. HTML defines a different set of options for each type of tag. The VCL reference for the *Custom* property of most Web items gives an example of possible options. For more information on possible options, use an HTML reference.

Customizing the InternetExpress page producer template

The template of an InternetExpress page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the page producer generates a default template as the value of the *HTMLDoc* property. This default template has the form

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

The HTML-transparent tags in the default template are translated as follows:

<#INCLUDES> generates the statements that include the javascript libraries. These statements have the form

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </SCRIPT>
```

<#STYLES> generates the statements that defines a style sheet from definitions listed in the *Styles* or *StylesFile* property of the InternetExpress page producer.

<#WARNINGS> generates nothing at runtime. At design time, it adds warning messages for problems detected while generating the HTML document. You can see these messages in the Web page editor.

<#FORMS> generates the HTML produced by the components that you add in the Web page editor. The HTML from each component is generated in the order it appears in *WebPageItems*.

<#SCRIPT> generates a block of javascript declarations that are used in the HTML generated by the components added in the Web page editor.

You can replace the default template by changing the value of *HTMLDoc* or setting the *HTMLFile* property. The customized HTML template can include any of the HTML-transparent tags that make up the default template. The InternetExpress page producer automatically translates these tags when you call the *Content* method. In addition, The InternetExpress page producer automatically translates three additional tags:

`<#BODYELEMENTS>` is replaced by the same HTML as results from the 5 tags in the default template. It is useful when generating a template in an HTML editor when you want to use the default layout but add additional elements using the editor.

`<#COMPONENT Name=WebComponentName>` is replaced by the HTML that the component named *WebComponentName* generates. This component can be one of the components added in the Web page editor, or it can be any component that supports the *IWebContent* interface and has the same Owner as the InternetExpress page producer.

`<#DATAPACKET XMLBroker=BrokerName>` is replaced with the XML data packet obtained from the XML broker specified by *BrokerName*. When, in the Web page editor, you see the HTML that the InternetExpress page producer generates, you see this tag instead of the actual XML data packet.

In addition, the customized template can include any other HTML-transparent tags that you define. When the InternetExpress page producer encounters a tag that is not one of the seven types it translates automatically, it generates an *OnHTMLTag* event, where you can write code to perform your own translations. For more information about HTML templates in general, see “HTML templates” on page 28-13.

Tip The components that appear in the Web page editor generate static code. That is, unless the application server changes the metadata that appears in data packets, the HTML is always the same, no matter when it is generated. You can avoid the overhead of generating this code dynamically at runtime in response to every request message by copying the generated HTML in the Web page editor and using it as a template. Because the Web page editor displays a <#DATAPACKET> tag instead of the actual XML, using this as a template still allows your application to fetch data packets from the application server dynamically.

Chapter

26

Using XML in database applications

In addition to the support for connecting to database servers, Delphi lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on Delphi's support for working directly with XML documents, see Chapter 30, "Working with XML documents."

Delphi's support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset) into XML documents and convert XML documents into data packets. In order to use these components, you must first define the transformation between the XML document and the data packet. Once you have defined the transformation, you can use special components to

- convert XML documents into data packets.
- provide data from and resolve updates to an XML document.
- use an XML document as the client of a provider.

Defining transformations

Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document. A description of this relationship is stored in a special XML document called a transformation.

Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or from the metadata in a data packet to an XML schema. Often, you create transformation files in pairs: one that maps from XML to data packet, and one that maps from data packet to XML.

In order to create the transformation files for a mapping, use the XMLMapper utility that ships in the bin directory.

Mapping between XML nodes and data packet fields

XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset.

Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):

```
<?xml version="1.0" standalone='yes' ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the new XML component support in Delphi.
        This looks like a good solution to our buisness-to-buisness application!
        Also attached, please find the project schedule. Do you think its reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

One natural mapping between this document and a dataset would map each email message to a single record. The record would have fields for the sender's name and address. Because an email message can have multiple recipients, the recipient (<to> would map to a nested dataset. Similarly, the cc list maps to a nested dataset. The subject line would map to a string field while the message itself (<content>) would probably be a memo field. The names of attachment files would map to a nested dataset because one message can have several attachments. Thus, the email above would map to a dataset something like the following:

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

where the nested dataset in the "To" field is

Name	Address
Joe Engineer	jengineer@MyCo.Com

the nested dataset in the "CC" field is

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

and the nested dataset in the "Attach" field is

Attachfile
XMLSpec.txt
Schedule.txt

Defining such a mapping involves identifying those nodes of the XML document that can be repeated and mapping them to nested datasets. Tagged elements that have values and appear only once (such as <content>...</content>) map to fields whose datatype reflects the type of data that can appear as the value. Attributes of a tag (such as the AttachFile attribute of the attachment tag) also map to fields.

Note that not all tags in the XML document appear in the corresponding dataset. For example, the <head>...<head/> element has no corresponding element in the resulting dataset. Typically, only elements that have values, elements that can be repeated, or the attributes of a tag map to the fields (including nested dataset fields) of a dataset. The exception to this rule is when a parent node in the XML document maps to a field whose value is built up from the values of the child nodes. For example, an XML document might contain a set of tags such as

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
</FullName>
```

which could map to a single dataset field with the value

```
Mr. John Smith
```

Using XMLMapper

The XML mapper utility, xmlmapper.exe, lets you define mappings in three ways:

- From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema.
- From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system.
- Between an existing XML schema and an existing data packet. This is useful when you have an XML schema and a database that both describe the same information and you want to make them work together.

Once you define the mapping, you can generate the transformation files that are used to convert XML documents to data packets and to convert data packets to XML documents. Note that only the transformation file is directional: a single mapping can be used to generate both the transformation from XML to data packet and from data packet to XML.

Note XML mapper relies on two .DLLs (midas.dll and msxml.dll) to work correctly. Be sure that you have both of these .DLLs installed before you try to use xmlmapper.exe. In addition, msxml.dll must be registered as a COM server. You can register it using Regsvr32.exe.

Loading an XML schema or data packet

Before you can define a mapping and generate a transformation file, you must first load descriptions of the XML document and the data packet between which you are mapping.

You can load an XML document or schema by choosing File|Open and selecting the document or schema in the resulting dialog.

You can load a data packet by choosing File|Open and selecting a data packet file in the resulting dialog. (The data packet is simply the file generated when you call a client dataset's *SaveToFile* method.) If you have not saved the data packet to disk, you can fetch the data packet directly from the application server of a multi-tiered application by right-clicking in the Datapacket pane and choosing Connect To Remote Server.

You can load only an XML document or schema, only a data packet, or you can load both. If you load only one side of the mapping, XML mapper can generate a natural mapping for the other side.

Defining mappings

The mapping between an XML document and a data packet need not include all of the fields in the data packet or all of the tagged elements in the XML document. Therefore, you must first specify those elements that are mapped. To specify these elements, first select the Mapping page in the central pane of the dialog.

To specify the elements of an XML document or schema that are mapped to fields in a data packet, select the Sample or Structure tab of the XML document pane and double-click on the nodes for elements that map to data packet fields.

To specify the fields of the data packet that are mapped to tagged elements or attributes in the XML document and double-click on the nodes for those fields in the Datapacket pane.

If you have only loaded one side of the mapping (the XML document or the data packet), you can generate the other side after you have selected the nodes that are mapped.

- If you are generating a data packet from an XML document, you first define attributes for the selected nodes that determine the types of fields to which they correspond in the data packet. In the center pane, select the Node Repository page. Select each node that participates in the mapping and indicate the attributes of the corresponding field. If the mapping is not straightforward (for example, a node with subnodes that corresponds to a field whose value is built from those subnodes), check the User Defined Translation check box. You will need to write an event handler later to perform the transformation on user defined nodes.

  Once you have specified the way nodes are to be mapped, choose Create | Datapacket from XML. The corresponding data packet is automatically generated and displayed in the Datapacket pane.

- If you are generating an XML document from a data packet, choose Create | XML from Datapacket. A dialog appears where you can specify the names of the tags and attributes in the XML document that correspond to fields, records, and datasets in the data packet. For field values, you specify whether they map to a tagged element with a value or to an attribute by the way you name them. Names that begin with an @ symbol map to attributes of the tag that corresponds to the record, while names that do not begin with an @ symbol map to tagged elements that have values and that are nested within the element for the record.
- If you have loaded both an XML document and a data packet (client dataset file), be sure you select corresponding nodes in the same order. The corresponding nodes should appear next to each other in the table at the top of the Mapping page.

Once you have loaded or generated both the XML document and the data packet and selected the nodes that appear in the mapping, the table at the top of the Mapping page should reflect the mapping you have defined.

Generating transformation files

To generate a transformation file, use the following steps:

1. First select the radio button that indicates what the transformation creates:
   - Choose the Datapacket to XML button if the mapping goes from data packet to XML document.
   - Choose the XML to Datapacket button if the mapping goes from XML document to data packet.

2 If you are generating a data packet, you will also want to use the radio buttons in the Create Datapacket As section. These buttons let you specify how the data packet will be used: as a dataset, as a delta packet for applying updates, or as the parameters to supply to a provider before fetching data.

3 Click Create and Test Transformation to generate an in-memory version of the transformation. XML mapper displays the XML document that would be generated for the data packet in the Datapacket pane or the data packet that would be generated for the XML document in the XML Document pane.

4 Finally, choose File | Save | Transformation to save the transformation file. The transformation file is a special XML file (with the .xtr extension) that describes the transformation you have defined.

Converting XML documents into data packets

Once you have created a transformation file that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application.

The *TXMLTransform* component transforms an XML document into a data packet according to the mapping in a transformation file.

Note You can also use *TXMLTransform* to convert a data packet that appears in XML format into an arbitrary XML document.

Specifying the source XML document

There are three ways to specify the source XML document:

- If the source document is a .xml file on disk, you can use the *SourceXmlFile* property.
- If the source document is an in-memory string of XML, you can use the *SourceXml* property.
- If you have an IDOMDocument interface for the source document, you can use the *SourceXmlDocument* property.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *SourceXmlFile* property. Only if *SourceXmlFile* is an empty string does it check the *SourceXml* property. Only if *SourceXml* is an empty string does it then check the *SourceXmlDocument* property.

Specifying the transformation

There are two ways to specify the transformation that converts the XML document into a data packet:

- Set the *TransformationFile* property to indicate a transformation file that was created using xmlmapper.exe.
- Set the *TransformationDocument* property if you have an *IDOMDocument* interface for the transformation.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *TransformationFile* property. Only if *TransformationFile* is an empty string does it check the *TransformationDocument* property.

Obtaining the resulting data packet

To cause *TXMLTranform* to perform its transformation and generate a data packet, you need only read the *Data* property. For example, the following code uses an XML document and transformation file to generate a data packet, which is then assigned to a client dataset:

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
```

Converting user-defined nodes

When you define a transformation using xmlmapper.exe, you can specify that some of the nodes in the XML document are “user-defined”. User-defined nodes are nodes for which you want to provide the transformation in code rather than relying on a straightforward node-value-to-field-value translation.

You can provide the code to translate user-defined nodes using the *OnTranslate* event. *OnTranslate* is called every time the *TXMLTransform* component encounters a user-defined node in the XML document. In the OnTranslate event handler, you can read the source document and specify the resulting value for the field in the data packet.

For example, the following *OnTranslate* event handler converts a node in the XML document with the following form

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

into a single field value:

```
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
```

```
    CurNode: IDOMNode;
  begin
    if Id = 'FullName' then
    begin
      Value = '';
      if SrcNode.hasChildNodes then
      begin
        CurNode := SrcNode.firstChild;
        Value := Value + CurNode.nodeValue;
        while CurNode <> SrcNode.lastChild do
        begin
          CurNode := CurNode.nextSibling;
          Value := Value + ' ';
          Value := Value + CurNode.nodeValue;
        end;
      end;
    end;
  end;
```

Using an XML document as the source for a provider

The *TXMLTransformProvider* component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Chapter 24, “Using provider components.” For information on using provider components with client datasets, see “Using a client dataset with a provider” on page 23-23.

You can specify the XML document from which the XML provider provides data and to which it applies updates using the *XMLDataFile* property.

TXMLTransformProvider components use internal *TXMLTransform* components to translate between data packets and the source XML document: one to translate the XML document into data packets, and one to translate data packets back into the XML format of the source document after applying updates. These two *TXMLTransform* components can be accessed using the *TransformRead* and *TransformWrite* properties, respectively.

When using *TXMLTransformProvider*, you must specify the transformations that these two *TXMLTransform* components use to translate between data packets and the source XML document. You do this by setting the *TXMLTransform* component’s *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component.

In addition, if the transformation includes any user-defined nodes, you must supply an *OnTranslate* event handler to the internal *TXMLTransform* components.

You do not need to specify the source document on the *TXMLTransform* components that are the values of *TransformRead* and *TransformWrite*. For *TransformRead*, the source is the file specified by the provider’s *XMLDataFile* property (although, if you set *XMLDataFile* to an empty string, you can supply the source document using *TransformRead.XmlSource* or *TransformRead.XmlSourceDocument*). For *TransformWrite*, the source is generated internally by the provider when it applies updates.

Using an XML document as the client of a provider

The *TXMLTransformClient* component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransform* client lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents.

To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the *ProviderName* property. As with the *ProviderName* property of a client dataset, *ProviderName* can be the name of a provider on a remote application server or it can be a local provider in the same form or data module as the *TXMLTransformClient* object. For information about providers, see Chapter 24, "Using provider components."

If the provider is on a remote application server, you must use a DataSnap connection component to connect to that application server. Specify the connection component using the *RemoteServer* property. For information on DataSnap connection components, see "Connecting to the application server" on page 25-23.

Fetching an XML document from a provider

TXMLTransformClient uses an internal *TXMLTransform* component to translate data packets from the provider into an XML document. You can access this TXMLTransform component as the value of the *TransformGetData* property.

Before you can create an XML document that represents the data from a provider, you must specify the transformation file that *TransformGetData* uses to translate the data packet into the appropriate XML format. You do this by setting the *TXMLTransform* component's *TransformationFile* or *TransformationDocument* property, just as when using a stand-alone *TXMLTransform* component. If that transformation includes any user-defined nodes, you will want to supply *TransformGetData* with an *OnTranslate* event handler as well.

There is no need to specify the source document for *TransformGetData*, *TXMLTransformClient* fetches that from the provider. However, if the provider expects any input parameters, you may want to set them before fetching the data. Use the *SetParams* method to supply these input parameters before you fetch data from the provider. *SetParams* takes two arguments: a string of XML from which to extract parameter values, and the name of a transformation file to translate that XML into a data packet. *SetParams* uses the transformation file to convert the string of XML into a data packet, and then extracts the parameter values from that data packet.

Note You can override either of these arguments if you want to specify the parameter document or transformation in another way. Simply set one of the properties on *TransformSetParams* property to indicate the document that contains the parameters or the transformation to use when converting them, and then set the argument you want to override to an empty string when you call *SetParams*. For details on the properties you can use, see “Converting XML documents into data packets” on page 26-6.

Once you have configured *TransformGetData* and supplied any input parameters, you can call the *GetDataAsXml* method to fetch the XML. *GetDataAsXml* sends the current parameter values to the provider, fetches a data packet, converts it into an XML document, and returns that document as a string. You can save this string to a file:

```
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
  XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
  XMLTransformClient1.TransFormSetParams.SourceXmlFile := 'InputParams.xml';
  XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
  XML := XMLTransformClient1.GetDataAsXml;
  XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
  try
    XMLDoc.Write(XML, Length(XML));
  finally
    XMLDoc.Free;
  end;
end;
```

Applying updates from an XML document to a provider

TXMLTransformClient also lets you insert all of the data from an XML document into the provider’s dataset or to delete all of the records in an XML document from the provider’s dataset. To perform these updates, call the *ApplyUpdates* method, passing in

- A string whose value is the contents of the XML document with the data to insert or delete.
- The name of a transformation file that can convert that XML data into an insert or delete delta packet. (When you define the transformation file using the XML mapper utility, you specify whether the transformation is for an insert or delete delta packet.)
- The number of update errors that can be tolerated before the update operation is aborted. If fewer than the specified number of records can’t be inserted or deleted, *ApplyUpdates* returns the number of actual failures. If more than the specified number of records can’t be inserted or deleted, the entire update operation is rolled back, and no update is performed.

The following call transforms the XML document Customers.xml into a delta packet and applies all updates regardless of the number of errors:

```
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
```

Part

III

Writing Internet applications

The chapters in “Writing Internet applications” present concepts and skills necessary for building applications that are distributed over the Internet.

Note The components described in this section are not available in all editions of Delphi.

Chapter 27

Creating Internet applications

Web server applications extend the functionality and capability of existing Web servers. A Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Any operation that you can perform with a Delphi application can be incorporated into a Web server application.

Delphi provides two different architectures for developing Web server applications: Web Broker and WebSnap. Although these two architectures are different, WebSnap and Web Broker have many common elements. The WebSnap architecture acts as a superset of Web Broker. It provides additional components, and new features such as the WebSnap Surface Designer—which allows the content of a page to be displayed without the developer having to run the application. Applications developed with WebSnap can include Web Broker components, whereas applications developed with Web Broker cannot include WebSnap components.

This chapter describes the features of the Web Broker and WebSnap technologies and provides general information on Internet-based client/server applications.

About Web Broker and WebSnap

The first step in building a Web server application is choosing which architecture you want to use. Both approaches provide many of the same features, including

- Support for many types of Web server applications, including ISAPI, NSAPI, CGI, Win CGI, and Apache. These are described in "Types of Web server applications" on page 27-6.
- Multithreading support so that incoming client requests are handled on separate threads.
- Caching of Web modules for quicker responses.

However, each approach has certain advantages and disadvantages. The major differences between these two approaches are outlined in the following table:

Table 27.1 Web Broker versus WebSnap

Web Broker	WebSnap
Backward compatible	Although WebSnap applications can use any Web Broker components that produce content, the Web modules and dispatcher that contain these are new.
Available in cross-platform (CLX) applications.	At present, WebSnap is only available on Windows.
Only one Web module allowed in an application.	Multiple Web modules can partition the application into units, allowing multiple developers to work on the same project with fewer conflicts.
Only one Web dispatcher allowed in the application.	Multiple, special-purpose dispatchers handle different types of requests.
Specialized components for creating content include page producers, InternetExpress components, and Web Services components.	Supports all the content producers that can appear in Web broker applications, plus many others designed to let you quickly build complex data-driven Web pages.
No scripting support.	Support for server-side scripting (JScript or VBscript) allows HTML generation logic to be separated from the business logic.
No built-in support for named pages.	Named pages can be automatically retrieved by a page dispatcher and addressed from server-side scripts.
No session support.	Sessions store information about an end user that is needed for a short period of time. This can be used for such tasks as login/logout support.
Every request must be explicitly handled, using either an action item or an auto-dispatching component.	Dispatch components automatically respond to a variety of requests.
Only a few specialized components provide previews of the content they produce. Most development is not visual.	The WebSnap surface designer lets you build Web pages visually, and view the results at design time. Previews are available for all components.

For more information on Web Broker, see Chapter 28, “Using Web Broker.” For more information on WebSnap, see Chapter 29, “Using WebSnap.”

Terminology and standards

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications:

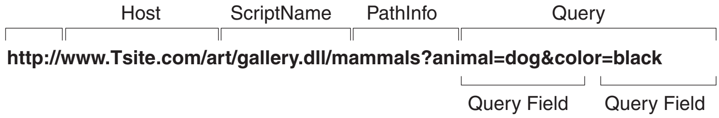
- RFC822, “Standard for the format of ARPA Internet text messages,” describes the structure and content of message headers.
- RFC1521, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies,” describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, “Hypertext Transfer Protocol — HTTP/1.0,” describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, `www.ietf.cnri.reston.va.us`

Parts of a Uniform Resource Locator

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in Figure 27.1:

Figure 27.1 Parts of a Uniform Resource Locator


Host
ScriptName
PathInfo
Query
http://www.Tsite.com/art/gallery.dll/mammals?animal=dog&color=black
Query Field
Query Field

The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on.

The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The ScriptName portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the pathinfo. This identifies the destination of the message within the Web server application. Path info values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The Query portion contains a set a named values. These values and their names are defined by the Web server application.

URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

HTTP request header information

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as “Host” followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case /art/gallery.dll/animals?animal=doc&color=black). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

HTTP server activity

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests.

Composing client requests

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Serving client requests

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application:

- If the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the content directly back to the server.
- If the program is WinCGI, the server opens a file and writes out the request information. It then executes the Win-CGI program, passing the location of the file containing the client information and the location of a file that the Win-CGI program should use to create content. The server waits while the program executes. When the program exits, the server reads the data from the content file written by the Win-CGI program.
- If the program is a dynamic-link library (DLL), the server loads the DLL (if necessary) and passes the information contained in the request to the DLL as a structure. The server waits while the program executes. When the DLL exits, it passes the content directly back to the server.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

Responding to client requests

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent also differs based on the type of program:

- When a Win-CGI script finishes it constructs a page of HTML, writes it to a file, writes any response information to another file, and passes the locations of both files back to the server. The server opens both files and passes the HTML page back to the client.

- When a DLL finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client.

Creating a Web server application as a DLL reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual request.

Types of Web server applications

Whether you use Web Broker or WebSnap, you can create five types of Web server applications. In addition, you can create a Web Application Debugger executable, which integrates the Web server into your application so that you can debug your application logic. The Web Application Debugger executable is intended only for debugging. When you deploy your application, you should migrate to one of the other five types.

Each of the five types of Web server application uses a type-specific descendant of *TWebApplication*, *TWebRequest*, and *TWebResponse*:

Table 27.2 Web server application components

Application Type	Application Object	Request Object	Response Object
Microsoft Server DLL (ISAPI)	*TISAPIApplication*	*TISAPIRequest*	*TISAPIResponse*
Netscape Server DLL (NSAPI)	*TISAPIApplication*	*TISAPIRequest*	*TISAPIResponse*
Apache Server DLL	*TApacheApplication*	*TApacheRequest*	*TApacheResponse*
Console CGI application	*TCGIApplication*	*TCGIRequest*	*TCGIResponse*
Windows CGI application	*TCGIApplication*	*TWinCGIRequest*	*TWinCGIResponse*

ISAPI and NSAPI

An ISAPI or NSAPI Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by *TISAPIApplication*, which creates *TISAPIRequest* and *TISAPIResponse* objects. Each request message is automatically handled in a separate execution thread.

Apache

An Apache Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by *TApacheApplication*, which creates *TApacheRequest* and *TApacheResponse* objects. Each request message is automatically handled in a separate execution thread.

CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by *TCGIApplication*, which creates

TCGIRequest and *TCGIResponse* objects. Each request message is handled by a separate instance of the application.

Win-CGI stand-alone

A Win-CGI stand-alone Web server application is a Windows application that receives client request information from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by *TCGIApplication*, which creates *TWinCGIRequest* and *TWinCGIResponse* objects. Each request message is handled by a separate instance of the application.

Debugging server applications

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting.

The following topics describe techniques you can use to debug Web server applications.

Using the Web Application Debugger

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application. This creates a Web server application that is also a COM server.

For information on how to write this Web server application using Web Broker, see Chapter 28, “Using Web Broker”. For more information on using WebSnap, see Chapter 29, “Using WebSnap”.

Launching your application with the Web Application Debugger

Once you have developed your Web server application, you can run and debug it as follows:

1. With your project loaded in the IDE, set any breakpoints so that you can debug your application just like any other executable.
2. Choose Run | Run. This displays the console window of the COM server that is your Web server application. The first time you run your application, it registers your COM server so that the Web App debugger can access it.

3 Select Tools | Web App Debugger.

4 Click the Start button. This displays the ServerInfo page in your default Browser.

5 The ServerInfo page provides a drop-down list of all registered Web Application Debugger executables. Select your application from the drop-down list. If you do not find your application in this drop-down list, try running your application as an executable. Your application must be run once so that it can register itself. If you still do not find your application in the drop-down list, try refreshing the Web page. (Sometimes the Web browser caches this page, preventing you from seeing the most recent changes.)

6 Once you have selected your application in the drop-down list, press the Go button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

Converting your application to another type of Web server application

When you have finished debugging your Web server application, you will need to convert it to another type of Web application before you install it with a commercial Web server. The following steps describe how to make these changes:

1 In the IDE, choose Project | Add New Project. This opens a new project without closing down the current one (you Web server application).

2 Launch the wizard to start a Web Broker or WebSnap application and choose the type of application you want to create.

3 Choose View | Project Manager to display the project manager.

4 In the project manager, drag all the units that make up your Web Server application from the old project to the one you just added. Omit the unit that implements the console window.

You now have a Web server application of the appropriate type.

Debugging Web applications that are DLLs

ISAPI , NSAPI, and Apache applications are actually DLLs that contain predefined entry points. The Web server passes request messages to the application by making calls to these entry points. Because these applications are DLLs, you can debug them by setting your application's run parameters to launch the server.

To set up your application's run parameters, choose Run | Parameters and set the Host Application and Run Parameters to specify the executable for the Web server and any parameters it requires when you launch it. For details about these values on your Web server, see the documentation provided by you Web server vendor.

Note Some Web Servers require additional changes before you have the rights to launch the Host Application in this way. See your Web server vendor for details.

Tip If you are using Windows 2000 with IIS 5, details on all of the changes you need to make to set up your rights properly are described at the following Web site:

http://community.borland.com/article/0,1410,23024,00.html

Once you have set the Host Application and Run Parameters, you can set up your breakpoints so that when the server passes a request message to your DLL, you hit one of your breakpoints, and can debug normally.

Note Before launching the Web server using your application's run parameters, make sure that the server is not already running.

Debugging under Windows NT

Under Windows NT, you must have the correct user rights to debug a DLL. In the User Manager, add your name to the lists granting rights for

- Log on as Service
- Act as part of the operation system
- Generate security audits

Tip Insert a hard-coded _asm int 3 into your code where you wish to begin debugging. Recreate your DLL and use Just-In-Time Debugging (Tools | Options | Debug).

Debugging under Windows 2000

Under Windows 2000, you grant the same rights as follows:

1. In the Administrative Tools portion of the Control Panel, click on Local Security Policy. Expand Local Policies and click on User Rights Assignment. Double-click on Act as part of the operating system in the right-hand panel.
2. Select Add to add a user to the list. Add your current user.
3. Reboot so the changes take effect.

Chapter

28

Using Web Broker

Web Broker components (located on the Internet tab of the component palette) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, You can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections.

The following sections of this chapter explain how you use the Web Broker components to create a Web server application.

Creating Web server applications with Web Broker

To create a new Web server application using the Web Broker architecture:

1 Select File | New | Other.

2 In the New Items dialog box, select the New tab and choose Web Server Application.

3 A dialog box appears, where you can select one of the Web server application types:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- Apache: Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Apache Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.

- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application, and adds the required entries to the uses clause of the project file.
- Web Application Debugger stand-alone executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components and containing an empty Web Module.

The Web module

The Web module (*TWebModule*) is a descendant of *TDataModule* and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer* and *TInetXPageProducer*, or descendants of *TCustomContentProducer* that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a multi-tiered database application.

In addition to storing non-visual components and business rules, the Web module also acts as a dispatcher, matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a *TWebDispatcher* component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from *TCustomWebDispatcher*, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the *TWebDispatcher* component that you add to a data module that replaces the Web module. If a second data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

Note The Web module that you set up at design time is actually a template. In ISAPI and NSAPI applications, each request message spawns a separate thread, and separate instances of the Web module and its contents are created dynamically for each thread.

Warning The Web module in a DLL-based Web server application is cached for later reuse to increase response time. The state of the dispatcher and its action list is not reinitialized between requests. Enabling or disabling action items during execution may cause unexpected results when that module is used for subsequent client requests.

The Web Application object

The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* (either *TISAPIApplication* or *TCGIApplication*) that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

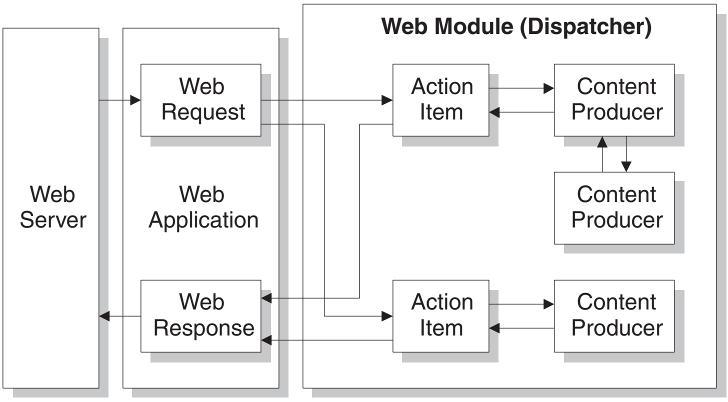
Warning Do not include the forms unit in the project **uses** clause after the CGIApp or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

The structure of a Web Broker application

When the Web application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

The Web dispatcher controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message. It is described more fully in the section “The Web dispatcher” on page 28-4.

Figure 28.1 Structure of a Server Application



The action items are responsible for reading the request and assembling a response message. Specialized content producer components aid the action items in dynamically generating the content of response messages, which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message. For more information on content producers, see "Generating the content of response messages" on page 28-13.

If you are creating the Web Client in a multi-tiered database application, your Web server application may include additional, auto-dispatching components that represent database information encoded in XML and database manipulation classes encoded in javascript. If you are creating a server that implements a Web Service, your Web server application may include an auto-dispatching component that passes SOAP-based messages on to an invoker that interprets and executes them. The dispatcher calls on these auto-dispatching components to handle the request message after it has tried all of its action items.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

The Web dispatcher

If you are using a Web module, it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (*TWebDispatcher*) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it chooses one or more action items to respond to the request.

Adding actions to the dispatcher

Open the action editor from the Object Inspector by clicking the ellipsis on the *Actions* property of the dispatcher. Action items can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

Action items are discussed in further detail in "Action items" on page 28-5.

Dispatching request messages

When the dispatcher receives the client request, it generates a *BeforeDispatch* event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items.

Next, the dispatcher looks through its list of action items for one that matches the pathinfo portion of the request message's target URL and that can provide the service specified as the method of the request message. It does this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

After checking all its action items, if the message is not handled the dispatcher checks any specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications, which are described in "Building Web applications using InternetExpress" on page 25-33

If, after checking all the action items and any specially registered auto-dispatching components, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an *AfterDispatch* event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

Action items

Each action item (*TWebActionItem*) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.

Determining when action items fire

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the Object Inspector and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the Object Inspector.

The target URL

The dispatcher compares the *PathInfo* property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

and assuming that the `/gallery.dll` part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

The request method type

The *MethodType* property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

Table 28.1 MethodType values

Value	Meaning
mtGet	The request is asking for the information associated with the target URI to be returned in a response message.
mtHead	The request is asking for the header properties of a response, as if servicing an *mtGet* request, but omitting the content of the response.
mtPost	The request is providing information to be posted to the Web application.
mtPut	The request asks that the resource associated with the target URI be replaced by the content of the request message.
mtAny	Matches any request method type, including *mtGet*, *mtHead*, *mtPut*, and *mtPost*.

Enabling and disabling action items

Each action item has an *Enabled* property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A *BeforeDispatch* event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

Caution Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. If the Web server application is a DLL that caches Web modules, the initial state will not be reinitialized for the next request. Use the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

Choosing a default action item

Only one of the action items can be the default action item. The default action item is selected by setting its *Default* property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the *Enabled* property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its *Enabled* property to *False*.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

Caution Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to *True*, that action will not be re-evaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

Responding to request messages with action items

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its *Producer* property, that producer automatically assigns the *Content* of the response message using its *Content* method. The Internet page of the component palette includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an *OnAction* event. The *OnAction* event handler is passed the *TWebRequest* object that represents the HTTP request message and a *TWebResponse* object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see "Generating the content of response messages" on page 28-13.

Sending the response

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

Using multiple action items

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to *False*.

If many action items divide up the work of responding to request messages, each setting *Handled* to *False* so that others can continue, make sure the default action item leaves the *Handled* parameter set to *True*. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

Accessing client request information

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of a *TWebRequest* object. In NSAPI and ISAPI applications, the request message is encapsulated by a *TISAPIRequest* object. Console CGI applications use *TCGIRequest* objects, and Windows CGI applications use *TWinCGIRequest* objects.

The properties of the request object are read-only. You can use them to gather all of the information available in the client request.

Properties that contain request header information

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the *GetFieldByName* method.

Properties that identify the target

The full target of the request message is given by the *URL* property. Usually, this is a URL that can be broken down into the protocol (HTTP), *Host* (server system), *ScriptName* (server application), *PathInfo* (location on the host), and *Query*.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the *QueryFields* property.

Properties that describe the Web client

The request also includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the *From* property), to the URI where the message originated (the *Referer* or *RemoteHost* property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the *DerivedFrom* property. You can also determine the IP address of the client (the *RemoteAddr* property), and the name and version of the application that sent the request (the *UserAgent* property).

Properties that identify the purpose of the request

The *Method* property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

Value	What the message requests
OPTIONS	Information about available communication options.
GET	Information identified by the *URL* property.
HEAD	Header information from an equivalent GET message, without the content of the response.
POST	The server application to post the data included in the *Content* property, as appropriate.
PUT	The server application to replace the resource indicated by the *URL* property with the data included in the *Content* property.
DELETE	The server application to delete or hide the resource identified by the *URL* property.
TRACE	The server application to send a loop-back to confirm receipt of the request.

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The *MethodType* property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

Properties that describe the expected response

The *Accept* property indicates the media types the Web client will accept as the content of the response message. The *IfModifiedSince* property specifies whether the client only wants information that has changed recently. The *Cookie* property includes state information (usually added previously by your application) that can modify the response.

Properties that describe the content

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the *ContentType* property, and its length in the *ContentLength* property. If the content of the message was encoded (for example, for data compression), this information is in the *ContentEncoding* property. The name and version number of the application that produced the content is specified by the *ContentVersion* property. The *Title* property may also provide information about the content.

The content of HTTP request messages

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database maintained by the Web server application.

The unprocessed value of the content is given by the *Content* property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the *ContentFields* property.

Creating HTTP response messages

When the Web server application creates a *TWebRequest* object for an incoming HTTP request message, it also creates a corresponding *TWebResponse* object to represent the response message that will be sent in return. In NSAPI and ISAPI applications, the response message is encapsulated by a *TISAPIResponse* object. Console CGI applications use *TCGIResponse* objects, and Windows CGI applications use *TWinCGIResponse* objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

Filling in the response header

Most of the properties of the *TWebResponse* object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

Indicating the response status

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the *StatusCode* property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the *ReasonString* property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

Indicating the need for client action

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the *Location* property. If the client must provide a password before you can proceed, set the *WWWAuthenticate* property.

Describing the server application

Some of the response header properties describe the capabilities of the Web server application. The *Allow* property indicates the methods to which the application can respond. The *Server* property gives the name and version number of the application used to generate the response. The *Cookies* property can hold state information about the client's use of the server application which is included in subsequent request messages.

Describing the content

Several properties describe the content of the response. *ContentType* gives the media type of the response, and *ContentVersion* is the version number for that media type. *ContentLength* gives the length of the response. If the content is encoded (such as for data compression), indicate this with the *ContentEncoding* property. If the content came from another URI, this should be indicated in the *DerivedFrom* property. If the value of the content is time-sensitive, the *LastModified* property and the *Expires* property indicate whether the value is still valid. The *Title* property can provide descriptive information about the content.

Setting the response content

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the *Content* property or the *ContentStream* property.

The *Content* property is a string. Delphi strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a *TFileStream* object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

Note If the value of the *ContentStream* property is not **nil**, the *Content* property is ignored.

Sending the response

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an *OnAction* event handler. The response object provides two methods for sending a response: *SendResponse* and *SendRedirect*. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

Generating the content of response messages

Delphi provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from *TCustomContentProducer* or one of its descendants.

TCustomContentProducer provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in the following section.
- Table producers create HTML commands based on the information in a dataset. They are described in “Using database information in responses” on page 28-17.

Using page producer components

Page producers (*TPageProducer* and its descendants) take an HTML template and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

HTML templates

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName=Value*, where there is no space between the parameter name, the equals symbol (=) and the value. The parameters are separated by whitespace.

The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the *TTag* data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

Tag Name	TTag value	What the tag should be converted to
Link	*tgLink*	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag.
Image	*tgImage*	A graphic image. The result is an HTML <IMG> tag.
Table	*tgTable*	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
ImageMap	*tgImageMap*	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.
Object	*tgObject*	An embedded ActiveX object. The result is an HTML sequence beginning with an <OBJECT> tag and ending with an </OBJECT> tag.
Embed	*tgEmbed*	A Netscape-compliant add-in DLL. The result is an HTML sequence beginning with an <EMBED> tag and ending with an </EMBED> tag.

Any other tag name is associated with *tgCustom*. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

Note The predefined tag names are case insensitive.

Specifying the HTML template

Page producers provide you with many choices in how to specify the HTML template. You can set the *HTMLFile* property to the name of a file that contains the HTML template. You can set the *HTMLDoc* property to a *TStrings* object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the *Content* method.

In addition, you can call the *ContentFromString* method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the *ContentFromStream* method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands, reading the template directly from a *TBlobStream* object.

Converting HTML-transparent tags

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the *OnHTMLTag* event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

Using page producers from an action item

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  PageProducer1.HTMLFile := 'Greeting.html';
  Response.Content := PageProducer1.Content;
end;
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (`<#UserName>`) in the HTML during execution:

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

If the content of the request message was the string *Mr. Ed*, the value of *Response.Content* would be

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our web site.
</BODY>
</HTML>
```

Note This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer’s *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item’s *Producer* property to accomplish the same effect as the *OnAction* event handler above.

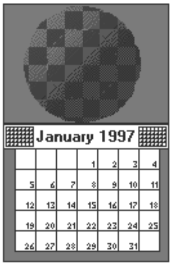
Chaining page producers together

The replacement text from an *OnHTMLTag* event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the *ContentFromString* method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:



January 1997

			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces <#MonthlyImage> with <#Image Month=January Year=1997>.
- Replaces <#TitleLine> with <#Calendar Month=December Year=1996 Size=Small> January 1997 <#Calendar Month=February Year=1997 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=January Year=1997 Size=Large>.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the <#Image Month=January Year=1997> tag with the appropriate HTML <IMG> tag. Yet another page producer resolves the #Calendar tags with appropriate HTML tables.

Using database information in responses

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

As an alternate approach, special components on the InternetExpress page of the component palette let you build Web servers that are part of a multi-tiered database application. See "Building Web applications using InternetExpress" on page 25-33 for details.

Adding a session to the Web module

Both console CGI applications and Win-CGI applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application. Each instance of the application has its own distinct, default session.

When writing an ISAPI application or an NSAPI application, however, each request message is handled in a separate thread of a single application instance. To prevent the database connections from different threads from interfering with each other, you must give each thread its own session.

Each request message in an ISAPI or NSAPI application spawns a new thread. The Web module for that thread is generated dynamically at runtime. Add a *TSession* object to the Web module to handle the database connections for the thread that contains the Web module.

Separate instances of the Web module are generated for each thread at runtime. Each of those modules contains the session object. Each of those sessions must have a separate name, so that the threads that handle separate request messages do not interfere with each other's database connections. To cause the session objects in each module to dynamically generate unique names for themselves, set the *AutoSessionName* property of the session object. Each session object will dynamically generate a unique name for itself and set the *SessionName* property of all datasets in the module to refer to that unique name. This allows all interaction with the database for each request thread to proceed without interfering with any of the other request messages. For more information on sessions, see "Managing database sessions" on page 20-16.

Representing database information in HTML

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer, which formats the fields of a dataset into the text of an HTML document.
- Table producers, which format the records of a dataset as an HTML table.

Using dataset page producers

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tagname which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see “Using page producer components” on page 28-13.

To use a dataset page producer, add a *TDataSetPageProducer* component to your web module and set its *DataSet* property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

Using table producers

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers, which format the fields of a dataset into the text of an HTML document.
- Query table producers, which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table’s color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

Specifying the table attributes

Table producers use the *THTMLTableAttributes* object to describe the visual appearance of the HTML table that displays the records from the dataset. The

THTMLTableAttributes object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the Object Inspector. Select the table producer object in the Object Inspector and expand the *TableAttributes* property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

Specifying the row attributes

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The *RowAttributes* property is a *THTMLTableRowAttributes* object.

At design time, specify these properties using the Object Inspector by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the *MaxRows* property.

Specifying the columns

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the Object Inspector after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate *THTMLTableColumn* objects and using the methods of the *Columns* property to add them to the table. If you do not set up the *Columns* property, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

Embedding tables in HTML documents

You can embed the HTML table that represents your dataset in a larger document by using the *Header* and *Footer* properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the *Caption* and *CaptionAlignment* properties to give your table a caption.

Setting up a dataset table producer

TDataSetTableProducer is a table producer that creates an HTML table for a dataset. Set the *DataSet* property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

Setting up a query table producer

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the *TQuery* object that uses those parameters as the *Query* property of a *TQueryTableProducer* component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the *Content* method of *TQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

Chapter

29

Using WebSnap

WebSnap augments WebBroker with new components, wizards, and views—making it easier to build Web applications that contain complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripts makes team development easier.

The dispatcher components automatically handle requests for page content, HTML form submissions, and requests for dynamic images. New components called adapters provide a means to define a scriptable interface to the business rules of your application. For example, the *TDataSetAdapter* object is used to make data-set components scriptable. You can use new producer components to quickly build complex, data-driven forms and tables, or to use XSL to generate a page. You can use the session component to keep track of end-users. You can use the user list component to provide access to user names, passwords, and access rights.

The Web application wizard allows you to quickly build an application that is customized with the components that you will need. The Web page module wizard allows you to create a module that defines a new page in your application. Or use the Web data module wizard to create a container for components that are shared across your Web application.

The page module views make it possible to see the result of server-side script without running the application. The Preview tab shows the page in an embedded browser. The HTML Result view shows the generated HTML. The XSL Tree and XML Tree views make it easier to work with XML and XSL.

To support a team of developers, you can use WebSnap's multiple-module support to partition the application into units that can be worked on independently. When creating a new page module, you can have the page module wizard create an external template file. You can edit the template file outside of the IDE and test it without recompiling the application.

The following sections of this chapter explain how you use the WebSnap components to create a Web server application.

Creating Web server applications with WebSnap

To create a new Web server application using the WebSnap architecture:

1 Select File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.

3 A dialog box appears which requires the following types of information:

   - Server type
   - Web application module types
   - Web application module options
   - Application components

Server type

Select one of the following types of Web server application, depending on your application's type of Web server:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Web server. It adds the library header to the project file, and adds the required entries to the uses list and to the exports clause of the project file.
- Apache: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Apache Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application and adds the required entries to the uses clause of the project file.
- Web Application Debugger executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web server application that communicates with the type of Web server your application will use.

Web application module types

The Web application module provides centralized control for business rules and non-visual components in the Web application. There are two types of Web application modules:

- Page Module: Selecting this type of module creates a content page. The page module contains a page producer which is responsible for generating the content of a page. The page producer displays its associated page when the HTTP request pathinfo matches the page name. The page can act as the default page when the pathinfo is blank.
- Data Module: Selecting this type of module does not create a content page. This module is used as a container for components shared by other modules—for example, database components used by two Web Page modules.

Web application module options

If the selected application module type is page module, you can associate a name with the page by entering a name in the Page Name field in the dialog box. At runtime, the instance of this module can be either kept in cache, or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options through the Page Options button. You can set the following categories:

- Producer: The producer type for the page can be set to one of *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, or *XSLPageProducer*. If the selected page producer supports scripting, then use the Script Engine drop-down list to select the language used to script the page.

Note The *AdapterPageProducer* supports only JScript.

- HTML: When the selected producer uses an HTML template this group will be visible.
- XSL: When the selected producer uses an XSL template, such as *TXSLPageProducer*, this group will be visible.
- New File: Check New File if you want a template file to be created and managed as part of the unit. A managed template file will appear in the project manager and have the same file name and location as the unit source file. Uncheck New File if you want to use the properties of the producer component (typically the *HTMLDoc* or *HTMLFile* property).
- Template: When New File is checked, choose the default content for the template file from the Template drop-down. The “Default” template displays the title of the application, the title of the page, and hyperlinks to published pages.
- Page: Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application's logic, whereas the title is the name that the end user will see when the page is displayed in a browser.

- Published: Check Published to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.
- Login Required: Check Login Required to require the user to log on before the page can be accessed.

Application components

Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page.

Selecting the Components button displays a dialog box where you can select one or more of the Application components:

- Application Adapter: Contains information about the application, such as the title. The default type is *TApplicationAdapter*.
- End User Adapter: Contains information about the user, such as their name, access rights, and whether they are logged in. The default type is *TEndUserAdapter*. *TEndUserSessionAdapter* may also be selected.
- Page Dispatcher: Examines the HTTP request's pathinfo, and calls the appropriate page module to return the content of a page. The default type is *TPageDispatcher*.
- Adapter Dispatcher: Automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components. The default type is *TAdapterDispatcher*.
- Dispatch Actions: Allows you to define a collection of action items to handle requests based on pathinfo and method type. Action items call user-defined events, or request the content of page-producer components. The default type is *TWebDispatcher*.
- Locate File Service: Provides control over the loading of template files, and script include files, when the Web application is running. The default type is *TLocateFileService*.
- Sessions Service: Used to store information about an end-users that is needed for a short period of time. For example, you can use sessions to keep track of logged-in users, and to automatically log a user out after a period of inactivity. The default type is *TSessionService*.
- User List Service: Keeps track of authorized users, and their passwords and access rights. The default type is *TWebUserList*.

For each of the above components, the component types listed are the default types shipped with the Delphi software product. Users can create their own component types or use third-party component types.

Web modules

There are four Web module types:

- *TWebAppPageModule*
- *TWebAppDataModule*
- *TWebPageModule*
- *TWebDataModule*

The Web application module (*TWebAppPageModule* or *TWebAppDataModule*) is a container for the application components that perform functions for the application as a whole—such as dispatching requests, managing sessions, and maintaining user lists. Your project can contain only one of these types of application modules.

The Web page module (*TWebPageModule)* provides content to a page and the Web data module (*TWebDataModule*) acts as a container for components shared across your application. You can optionally include one or more Web page and Web data modules in the Web application module.

Web data modules

Like standard data modules, a Web data module is a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. When the application is running, a data module does not create a window.

The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements.

Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both:

- a page module that displays a grid, and
- a page module that displays an input form.

Structure of a Web data module unit

Standard data modules have a variable called the form variable, which is used to access the data module object. Web data modules replace this with a function. The purpose is the same. However, because WebSnap applications may be multi-threaded and may have multiple instances of a particular module that service multiple requests concurrently, this function is implemented to return the correct instance.

The unit also registers a factory. The factory specifies how the module should be managed by the WebSnap application. For example, flags indicate whether to cache the module and reuse it for other requests, or to destroy the module after a request has been serviced.

Interfaces implemented by a Web data module

A Web data module implements the following interfaces:

INotifyWebActivate: This interface is called when the module is activated to service a Web request, and before the module is deactivated after a Web request is serviced.

IWebVariablesContainer: This interface is called during script execution to resolve variable references. The adapter dispatcher also calls this interface to locate adapter actions and fields that are referenced in an HTTP request.

IGetScriptObject: This interface is called to retrieve the module's IDispatch implementation. The returned object is the interface between the module and the active scripting engine.

IIteratorObjectSupport: This interface is used to iterate through all of the objects within the module that can be accessed by active scripting.

Web page modules

The page module has a page producer component associated with it. When a request is received, the page dispatcher analyses the request and calls the appropriate page module to process the request and return the content of the page.

Like Web data modules, Web page modules are containers for components. The difference between a Web data module and a Web page module is that a Web page module is used to produce a Web page.

Page producer component

Web page modules have a property that identifies the page producer component responsible for generating content for the page. The WebSnap wizards automatically adds a producer when creating a Web page module. You can change the page producer component later by dropping in a different producer from the WebSnap palette. However, if the page module has a template file, be sure that the content of this file is compatible with the producer component.

Page name

Web page modules have a page name that can be used to reference the page in an HTTP request or within the application's logic. A factory in the Web page module's unit specifies the page name for the Web page module.

Producer template

Most page producers use a template. HTML templates typically contain some static HTML mixed in with transparent tags or server-side script. When page producers create their content, they replace the transparent tags with appropriate values and execute the server-side script to produce the HTML that is displayed by a client browser. The XSLPageProducer is an exception to this. It uses XSL templates, which contain XSL rather than HTML. The XSL templates do not support transparent tags or server-side script.

Web page modules may have an associated template file that is managed as part of the unit. A managed template file appears in the project manager and has the same base file name and location as the unit service file. If the Web page module does not have an associated template file then the properties of the page producer component specify the template.

Interfaces that the Web page module implements

A Web page module implements all of the interfaces of a Web data module, in addition to the following:

IDefaultPageFileName: The WebSnap surface designer uses this interface to ensure that a page module uses the proper template file.

ISetWebContentOptions: The WebSnap surface designer uses this interface to control the content that the page module generates. For example, an option can be used to retrieve the content before the Active Script executes.

IGetProducerComponent: The WebSnap surface designer uses this interface to retrieve the producer component associated with the page module.

IProducerEditorViewSupport: The WebSnap surface designer uses this interface to retrieve information about the editor views that it should display when the page module is active. The available editor views include HTML Script, Preview, HTML Result, XSL Tree, and XML Tree.

IPageResult: This interface provides access to the result that a page module produces. This interface supports three types of results: HTTP Content, HTTP Redirection, and Page Include.

IGetDefaultAction: This interface retrieves the default adapter action (if any) that is associated with the page module.

Web application modules

The Web application module implements interfaces that are not implemented by *TWebPageModule* or *TWebDataModule*.

Interfaces implemented by a Web application data module

A Web application data module implements all the interfaces of a Web data module, in addition to the following:

IGetWebAppServices: Retrieves the interface to the application's Web request handler.

IGetWebAppComponents: Retrieves the interface to the application-level components, including the AdapterDispatcher, PageDispatcher, SessionsService, and EndUserAdapter.

Interfaces implemented by a Web application page module

A Web application page module implements all the interfaces of a Web Application data module and a Web page module.

Adapters

Adapters provide a way to create an interface to the application data. They allow you to insert scripting languages into a page, and to retrieve information by making calls from your script code to the adapters.

For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the values of those data fields.

Fields

Fields are components that the page producer uses to retrieve data from your application and to display the content on a Web page. Fields can also be used to retrieve an image. In this case, the field returns the address of the image written to the Web page. When a page displays its content, a request sent to the Web application, invokes the adapter dispatcher to retrieve the actual image from the field component.

Actions

Actions are components that execute commands on behalf of the adapter. When a page producer generates its page, the scripting language calls adapter action components to return the name of the action along with any parameters necessary to execute the command. For example, consider clicking a button on an HTML form to delete a row from a table. This returns, in the HTTP request, the action name associated with the button and a parameter indicating the row number. The adapter dispatcher locates the named action component and passes the row number as a parameter to the action.

Errors

Adapters keep a list of errors that occur while executing an action. Page producers can access this list of errors and display them in the Web page that the application returns to the end user.

Records

Some adapter components, such as *TDataSetAdapter*, represent multiple rows. The adapter provides a scripting interface which allows iteration through the rows. Some adapters support paging, and iterate over only the rows on the current page.

Page producers

You use page producers to generate content on behalf of a Web page module. You can also use producers in the same way as they are used in WebBroker applications, by associating the producer with a Web dispatcher action item. The advantages of using the Web page module are:

- the ability to preview the page’s layout without running the application, and
- the ability to associate a page name with the module, so that the page dispatcher can call the page producer automatically.

Templates

Producers provide the following functionality:

- They generate HTML content.
- They can reference an external file using the HTMLfile property, or the internal string using the HTMLDoc property.
- When the producers are used in conjunction with a Web page module, the template can be a file associated with a unit.
- Producers dynamically generate HTML which can be inserted into the template using transparent tags or active scripting. Transparent tags can be used in the same way as WebBroker applications. For more details, see “Converting HTML-transparent tags” on page 28-14. Active scripting support allows you to embed JavaScript or VBscript inside the HTML page.

Server-side scripting in WebSnap

Page producer templates can include scripting languages such as JScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser).

Active scripting

WebSnap relies on *active scripting* to implement server-side script. Active scripting is a technology created by Microsoft to allow a scripting language to be used with application objects through COM interfaces. Microsoft ships two active scripting languages, VBScript and JScript. Support for other languages is available through third parties.

Script engine

The page producer's *ScriptEngine* property identifies the active scripting engine that evaluates the script within a template.

Script blocks

Script blocks are delimited by <% and %>. The script engine evaluates any text inside script blocks. The result becomes part of the page producer's content. The page producer writes text outside of a script block after translating any embedded transparent tags. Script blocks can also enclose text, allowing conditional logic and loops to dictate the output of text. For example, the following JScript block generates a list of five numbered lines:

```
<ul>
<% for (i=0;i<5;i++) { %>
    <li>Item <% Response.Write(i) %></li>
<% } %>
</ul>
```

The following script block is equivalent:

```
<ul>
<% for (i=0;i<5;i++) { %>
    <li>Item <%=i %></li>
<% } %>
</ul>
```

The <%= delimiter is short for *Response.Write*.

Creating script

Developers can take advantage of WebSnap features to automatically generate script.

Wizard templates

When they create a new WebSnap application or page module, WebSnap wizards provide a template field that is used to select the initial content for the page module template. For example, the template called "Default" generates JScript to display the application title, page name, and links to published pages.

TAdapterPageProducer

The *TAdapterPageProducer* builds forms and tables by generating HTML and JScript. The generated JScript calls adapter objects to retrieve field values, field image parameters, and action parameters.

Editing and viewing script

The WebSnap surface designer provides a view of your Web Page modules which lets you preview a scripted page. Use the HTML Result tab to view the HTML resulting from the executed script. Use the Preview tab to view the result in a browser. The HTML Script tab is available when the Web Page module uses *TAdapterPageProducer*. The HTML Script tab displays the HTML and JScript generated by the *TAdapterPageProducer* object. Consult this view to see how to write script that builds HTML forms to display adapter fields and execute adapter actions.

Including script in a page

A template can include script from a file or from another page. To include script from a file, use the following code statement:

```
<!-- #include file="filename.html" -->
```

When the template includes script from another page, the script is evaluated by the including page, use the following code statement to include the unevaluated content of page1.

```
<!-- #include page="page1" -- >
```

Script objects

Script objects are either VCL or CLX objects that can be referenced by script. You make VCL or CLX objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting:

- **Application—**The application object (which may be null) provides access to the application adapter of the Web Application module. The following JScript block writes the application title:

  ```
  <%= Application.Title %>
  ```

- **EndUser—**The EndUser object provides access to the end-user adapter of the Web Application module. The following JScript block writes the end-user name:

  ```
  <%= EndUser.DisplayName %>
  ```

- **Session—**The session object provides access to the session object of the Web Application module. The following JScript block writes the session ID:

  ```
  <%= Session.SessionID %>
  ```

- **Pages—**The pages object (*Pages*) provides access to the application pages. The following JScript block writes links to all published pages:

  ```
  <%  e = new Enumerator(Pages)
       for (; !e.atEnd(); e.moveNext())
       {
         if (e.item().Published)
         {
             Response.Write('<A HREF="' + e.item().HREF +'">' + e.item().Title + '</A>')
         }
       }
  %>
  ```

Note that the editor's Preview tab will not display the proper result of this script block. The pages object is always empty at design time because the Web page module factories have not been registered.

- **Modules—**The modules object provides access to the application modules. The following JScript block writes the content of an adapter field in a module called DM.

```
<%= Modules.DM.Adapter1.Field1.DisplayText %>
```

- **Page—**The Page object provides access to the current page. The following JScript block writes the title of the current page:

```
<%= Page.Title %>
```

- **Producer—**The Producer object provides access to the page producer of the Web Page module. The following JScript block evaluates a transparent tag before writing the content:

```
<% Producer.Write('Here is a tag <#TAG>') %>
```

Note that the editor's Preview tab will probably not display the proper result of this script block. The event handlers that usually replace transparent tags do not execute unless the application is running.

- **Response—**The Response object provides access to the WebResponse. Use this object when tag replacement is not desired.

```
<% Response.Write('Hello World!') %>
```

- **Request—**The Request object provides access to the WebRequest. The following JScript block displays the pathinfo.

```
<%= Request.PathInfo %>
```

- **Adapter Objects—**All of the adapter's components on the current page can be referenced without qualification. Adapter's in other modules must be qualified using the Modules objects. The following JScript block displays the text value of the *FirstName* field from of all rows of Adapter1:

```
<% e = new Enumerator(Adapter1.Records) %>
  <% for (; !e.atEnd(); e.moveNext()) %>
  <% { %>
      <p><%= Adapter1.FirstName.DisplayText %>
  <% } %>
```

Dispatching requests

When the WebSnap application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned.

WebContext

Before handling the request, the Web application module initializes the WebContext object (of type *TWebContext*). The WebContext is a thread variable that provides global access to variables used by components servicing the request. For example, the WebContext contains the *TWebResponse* and *TWebRequest* objects, as well as the adapter request and adapter response objects discussed later in this section.

Dispatcher components

The dispatcher components within the Web Application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request.

The adapter dispatcher component (*TAdapterDispatcher)* looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it will pass control to that component.

The Web Dispatcher component (*TWebDispatcher*) maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The Web dispatcher looks for an action item that matches the request. If it finds one, it passes control to that action item. The Web dispatcher also looks for auto-dispatching components that can handle the request.

The page dispatcher component (*TPageDispatcher*) examines the pathInfo property of the *TWebRequest* object, looking for the name of a registered Web page module. If the dispatcher finds a Web page module name, then it will pass control to that Web page module.

Adapter dispatcher operation

The adapter dispatcher component automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

Using adapter components to generate content

In order for WebSnap applications to automatically execute adapter actions and retrieve dynamic images from adapter fields, the HTML content must be properly constructed. If the HTML content is not properly constructed, then the resulting HTTP request will not contain the information that the adapter dispatcher needs to call adapter action and field components.

To reduce errors in constructing the HTML page, adapter components indicate what the names and values of HTML elements must be. Adapter components have methods that retrieve the names and values of hidden fields that must appear on an HTML form used to update adapter fields. Typically, page producers use server-side scripts to retrieve names and values from adapter components and generates HTML using these names and values. For example, the following script constructs an <IMG> element that references the field called Graphic from Adapter1:

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

When the Web application evaluates the script, the HTML src attribute will contain the information necessary to identify the field and any parameters that the field component needs to retrieve the image. The resulting HTML might look like this:

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

When the browser sends an HTTP request to retrieve this image to the Web application, the adapter dispatcher will be able to determine that the Graphic field of Adapter1, in the module DM, should be called with the parameter "Species No=90090". The adapter dispatcher will call the Graphic field to write an appropriate HTTP response.

The following script constructs an <A> element referencing the EditRow action of Adapter1 and the page called "Details":

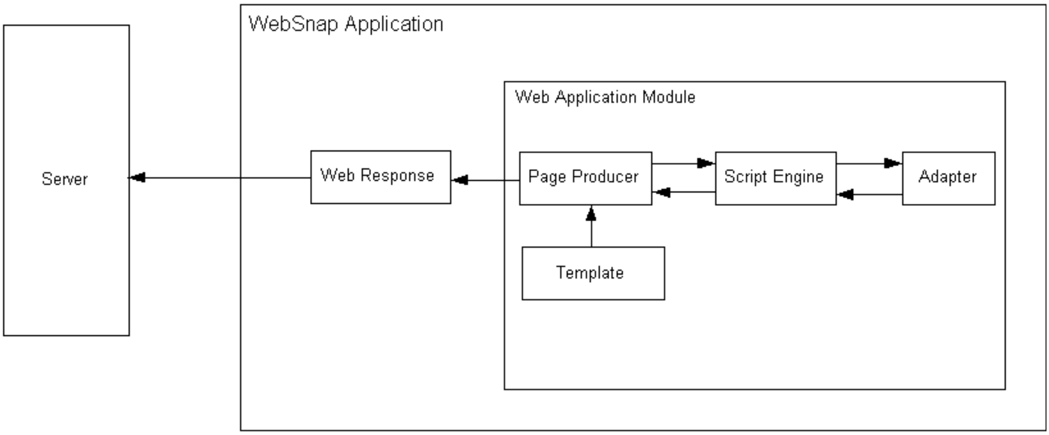
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

The resulting HTML might look like this:

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

When the end-user clicks this hyperlink and the browser sends an HTTP request, the adapter dispatcher will be able to determine that the EditRow action of Adapter1, in the module DM, should be called with the parameter "Species No=903010". The adapter dispatcher will also indicate that the Edit page is to be displayed if the action executes successfully, and that the Grid page is to be displayed if action execution fails. It will then call the EditRow action to locate the row to be edited, and the page named Edit will be called to generate an HTTP response. Figure 29.1 shows how adapter components are used to generate content.

Figure 29.1 Generating Content Flow



Adapter requests and responses

When the adapter dispatcher receives the client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about the HTTP request. The adapter request and adapter response objects are stored in the WebContext to allow access during the processing of the request.

The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field.

The adapter response object is used by the adapter component to indicate the response to an adapter action or adapter image request. There are two types of adapter response objects, action and image.

Action request

The action request object is responsible for breaking the HTTP request down into information needed to execute an adapter action. The types of information needed for executing an adapter action may include:

- **Component Name—**Identifies the adapter action component.
- **Adapter Mode—**Adapters can define a mode. For example, *TDataSetAdapter* supports Edit, Insert, and Browse modes. An adapter action may execute differently depending on the mode. For example, the *TDataSetAdapter* Apply action adds a new record when in Insert mode, and updates a record when in Edit mode.
- **Success Page—**The success page identifies the page displayed after successful execution of the action.
- **Failure Page—**The failure page identifies the page displayed if an error occurs during execution of the action.
- **Action Request Parameters—**This identifies the parameters need by the adapter action. For example, the *TDataSetAdapter* Apply action will include the key values identifying the record to be updated.
- **Adapter Field Values—**These are the values for the adapter fields passed in the HTTP request when an HTML form is submitted. A field value can include new values entered by the end-user, the original values of the adapter field, and uploaded files.
- **Record Keys—**If an HTML form submits changes to multiple records, keys used by the adapter action component are required to uniquely identify each record so that the adapter action can be performed on each record. For example, when the *TDataSetAdapter* Apply action is performed on multiple records, the record keys are used to locate each record in the dataset before updating the dataset fields.

Action response

The Action Response object generates an HTTP response on behalf of an adapter action component. The adapter action indicates the type of response by setting properties within the object, or by calling methods in the Action Response object. The properties include:

- *Redirect Options*—The redirect options indicate whether to perform an HTTP redirect instead of returning HTML content.
- *Execution Status*—Setting the status to success causes the default action response to be the content of the success page identified in the Action Request.

The Action response methods include:

- *RespondWithPag* —The adapter action calls this method when a particular Web page module should generate the response.
- *RespondWithComponent*—The adapter action calls this method when the response should come from the Web page module containing this component.
- *RespondWithURL*—The adapter action calls this method when the response is a redirect to a specified URL.

When responding with a page, the Action response object attempts to use the page dispatcher to generate page content. If it does not find the page dispatcher, it calls the Web Page module directly.

Figure 29.4 illustrates how action request and action response objects handle a request.

Figure 29.2 Action request and response

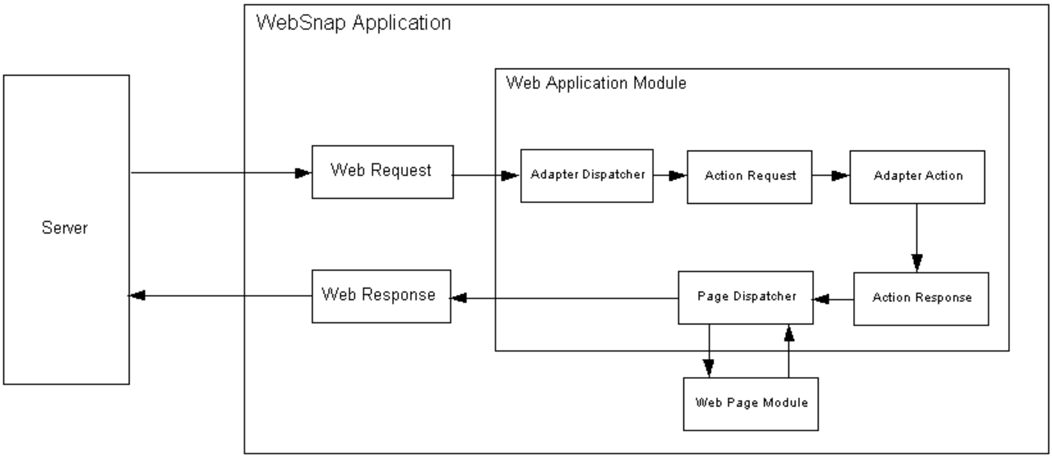


Image request

The Image Request object is responsible for breaking the HTTP request down into the information required by the adapter image field to generate an image. The types of information represented by the Image Request include:

- Component Name - Identifies the adapter field component.

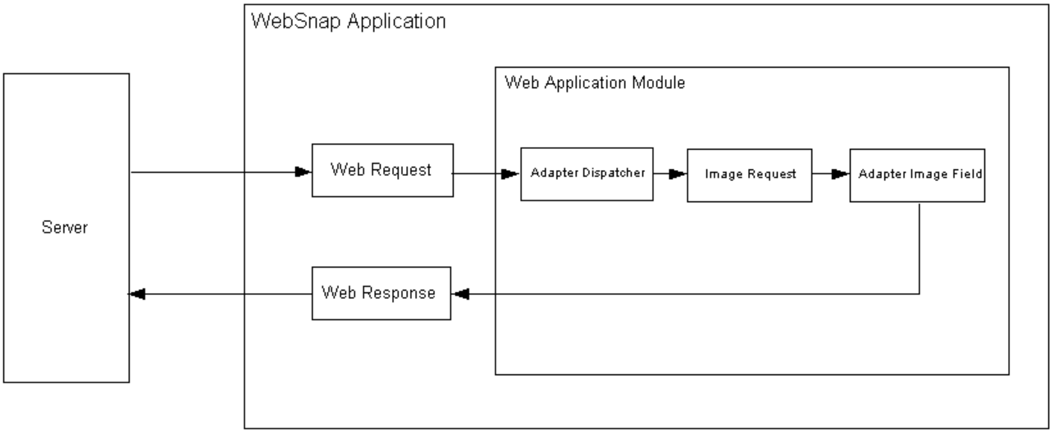
- Image Request Parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

Image response

The Image response object contains the *TWebResponse* object. Adapter fields respond to an adapter request by writing an image to the Web response object.

Figure 29.3 illustrates how adapter image fields respond to a request.

Figure 29.3 Image response to a request



Dispatching action items

The Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that:

- matches the Pathinfo portion of the target URL's request message, and
- can provide the service specified as the method of the request message.

It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response, or signals that the request has been completely handled.
- Adds to the response, and then allows other action items to complete the job.
- Defers the request to other action items.

After the dispatcher has checked all of its action items, if the message has not been handled correctly, the dispatcher checks for specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications. If the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

Page dispatcher operation

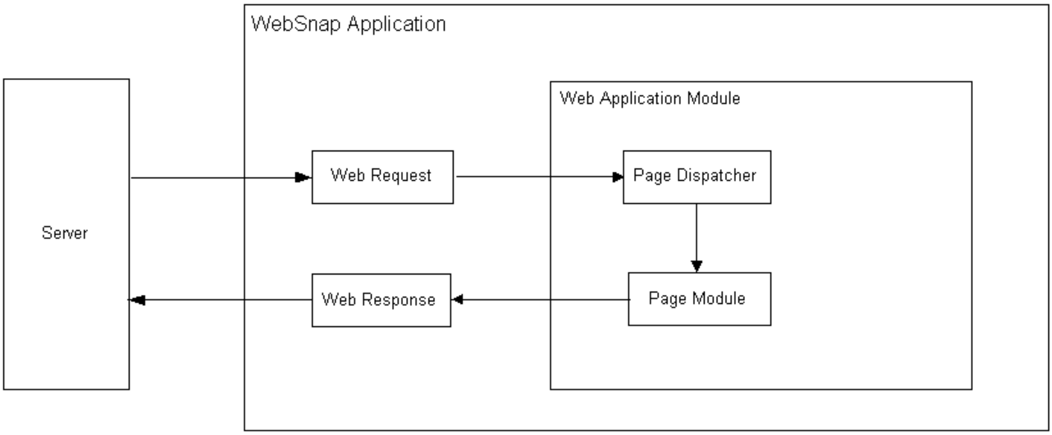
When the page dispatcher receives a client request, it determines the page name by checking the Pathinfo portion of the target URL's request message. If the pathinfo portion is not blank, the page dispatcher uses the ending word of pathinfo as the page name. If the pathinfo portion is blank, the page dispatcher tries to determine a default page name.

If the page dispatcher's *DefaultPage* property contains a page name, then the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank, and the Web application module is a page module, then the page dispatcher uses the name of the Web application module as the default page name.

If the page name is not blank, the page dispatcher searches for a Web page module with a matching name. If it finds a Web page module, then it calls that module to generated a response. If the page name is blank, or if the page dispatcher does not find a Web page module, the page dispatcher raises an exception.

Figure 29.4 shows how the page dispatcher responds to a request.

Figure 29.4 Dispatching a page



WebSnap tutorial

The following section describes the steps to build a WebSnap application. Completing the tutorial will familiarize users with the WebSnap architecture and new concepts, by incorporating the new dispatcher and adapter components into the new Web Page module. The WebSnap application demonstrates how to use WebSnap HTML components to build an application that edits the content of a table.

Create a new application

To create a new WebSnap application:

Step 1. Start the WebSnap application wizard

1 Run the Delphi application and select File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.

3 In the New WebSnap Application dialog box:

- Select the Web App Debugger Executable.
- In the CoClass field type **CountryTutorial.**
- Select Page Module as the component type.
- In the Page Name field type **Home.**

4 Click OK.

Step 2. Save the generated files and project

To save the pascal unit file and project:

1 Select File | Save All.

2 In the File name field enter **HomeU.pas** and Click OK.

3 In the File name field enter **CountryU.pas** and Click OK.

4 In the File name field enter **CountryTutorial.dpr** and Click OK.

5 Click OK.

Note Save the unit and the project to the same directory since the application will look for the HomeU.html file in the same directory as the executable.

Step 3. Specify the application title

The application title is the name displayed to the end user. To specify the application title:

1 Select View | Project Manager.

2 In the Project Manager window expand CountryTutorial.exe and double click the HomeU entry.

3 In the Object Inspector window (bottom left), select ApplicationAdapter from the pull down list.

4 In the Properties tab, enter **Country Tutorial** in the ApplicationTitle field.

5 Click on the Preview tab in the editor window. The application title is displayed at the top of the page.

Create a CountryTable page

A Web page module is used to define a published page, and as a container for data components.

Step 1. Add a new module

To add a new module:

1 Select File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.

3 In the dialog box, set the Producer Type to AdapterPageProducer from the list.

4 In the Page Name field enter **CountryTable.**

5 Leave the rest of the fields and selection at their default values.

6 Click OK.

Step 2. Save the new module

Save the unit to the directory of the project file. When the application runs, it searches for the CountryTableU.html file in the same directory as the executable.

1 Select File | Save.

2 In the File name field, enter **CountryTableU.pas** and Click OK.

Add data components to the CountryTable module

A *TTable* component provides the data for the HTML table. The *TDataSetAdapter* component allows server side script to access the *TTable* component.

Step 1. Add data-aware components

1 Select View | Project Manager.

2 In the Project Manager window expand CountryTutorial.exe and double click the CountryTableU entry.

3 Select View | Object TreeView. The Object TreeView window (left hand side) becomes active.

4 Select the Data Access tab in the tool palette.

5 Select a Table component (left-click and hold) and drag the component to the Object TreeView window.

6 Select a Session component (left-click and hold) and drag the component to the Object TreeView window.

7 Select the Session component in the Object TreeView window. This displays the Session component values in the Object Inspector window.

8 In the Object Inspector window, set the AutoSessionName property to *True*.

9 Select the Table component in the Object TreeView window. This displays the Table component values in the Object Inspector window.

10 In the Object Inspector window, change the following properties:

- Set the *Active* property to *True*.
- Set the *DatabaseName* property to DBDEMOS .
- In the *Name* property, type **Country**.
- Set the *TableName* property to country.db.

Note The Session component is required because we are using the BDE component (*TTable*) in a multi threaded application.

Step 2. Specify a key field

The key field is used to identify records within a table. This becomes important when we add an edit page to the application. To specify a key field:

1 In the Object Tree View window, expand the Session and DBDemos node, and select the country.db node. This node is the Country Table component.

2 Right-click on the country.db node and select Fields Editor.

3 Right-click in the CountryTable.Country editor window and select the Add All Fields command.

4 Select the Name field from the list of added fields.

5 In the Object Inspector window, expand the *ProviderFlags* property.

6 Set the pfInKey property value to *True*.

Step 3. Add an adapter component

To expose the data in the *TTable* server-side scripting, you must include a DataSetAdapter (*TDataSetAdapter*) component. To add such a component:

1 Select the WebSnap tab in the tool palette.

2 Select the DataSetAdapter component (left-click and hold) and drag the component to the Object TreeView window.

3 In the Object Inspector window, change the following properties:

- Set the DataSet field to Country.
- In the Name field type **Adapter**.

Create a grid to display the data

The *AdapterPageProducer* leverages server-side script to quickly build an HTML table.

Step 1. Add a grid

To add a grid to display the data from the Country table:

1 Select View | Project Manager.

2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryTableU entry.

3 Select View | Object TreeView. The Object TreeView window (left-hand side) becomes active.

4 Expand the *AdapterPageProducer* component.

5 Right-click on WebPageItems entry and select New Component.

6 In the Add Web Component window, select AdapterForm, then Click OK. An AdapterForm1 component appears in the Object TreeView window.

7 Right-click on AdapterForm1 and select New Component.

8 In the Add Web Component window, select AdapterGrid then click OK. An AdapterGrid1 component appears in the Object TreeView window.

9 In the Object Inspector window, set the Adapter property to Adapter.

10 To preview the Page, select the CountryTableU.pas tab in the code editor window, and select the Preview tab at the bottom. If the Preview tab is not shown, use the right arrow at the bottom to scroll through the tabs.

11 Select the HTML Script tab to view the JScript generated by the WebSnap components.

Step 2. Add editing commands to the grid

Users may need to update the content of the table. To allow users to make such updates, such as deleting or inserting a row, add command components.

To add command components:

1 In the Object TreeView window for the CountryTable, expand the *AdapterPageProducer* component and all its branches.

2 Right-click on the AdapterGrid1 component and select Add All Columns.

3 Right-click on the AdapterGrid1 component and select New Component. An AdapterCommandColumn1 entry is added to the AdapterGrid1 component.

4 Right-click on AdapterCommandColumn1 and choose Add Commands.

5 Multi-select the DeleteRow, EditRow, and NewRow commands; then click OK.

6 To preview the Page, click on the Preview tab at the bottom of the code editor.

Add an edit form

Create a Web page module to be the Edit form for the country table.

Step 1. Add a new module

To add a new WebSnap page module:

1 Select File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.

3 In the dialog box, set the Producer Type to AdapterPageProducer from the list.

4 In the Page Name field, enter **CountryForm**.

5 Uncheck the Published box.

6 Leave the rest of the fields and selections at their default values.

7 Click OK.

Step 2. Save the new module

Save the unit to the directory as the project file. When the application runs, it will look for the CountryFormU.html file in the same directory as the executable.

1 Select File | Save.

2 In the File name field enter **CountryFormU.pas** and Click OK.

Step 3. Use the CountryTableU unit

Add CountryTableU unit to the **uses** clause to allow the module access to the Adapter component.

1 Select File | Use Unit.

2 Select CountryTableU from the list then click OK.

Step 4. Add input fields

Add components to the *AdapterPageProducer* component to generate data entry fields in the HTML form.

To add input fields:

1 Select View | Project Manager.

2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryFormU entry.

3 Select View | Object TreeView. The Object TreeView window (left-hand side) becomes active.

4 In the Object TreeView window, expand the *AdapterPageProducer* component, right-click on WebPageItems, and select New Component.

5 Select AdapterForm, then click OK. An AdapterForm1 entry appears in the Object TreeView window.

6 Right-click on AdapterForm1 and select New Component.

7 Select AdapterFieldGroup then click OK. An AdapterFieldGroup1 entry appears in the Object TreeView window.

8 In the Object Inspector window, set the Adapter property to CountryTable.Adapter.

9 To preview the Page, click the Preview tab at the bottom of the code editor.

Step 5. Add buttons

Add components to the *AdapterPageProducer* component to generate the submit buttons in the HTML form. To add components:

1 In the Object TreeView, expand the *AdapterPageProducer* component and all its branches.

2 Right-click on AdapterForm1 entry and select New Component.

3 Select AdapterCommandGroup then click OK. An AdapterCommandGroup1 entry appears in the Object TreeView window.

4 In the Object Inspector window, set the DisplayComponent property to AdapterFieldGroup1.

5 Right-click on AdapterCommandGroup1 entry and select Add Commands.

6 Multi-select the Cancel, Apply, and Refresh Row commands; then click OK.

7 To preview the Page, click the Preview tab at the bottom of the code editor window. If the preview does not show the country form, click on the Code tab and then re-click the Preview tab.

Step 6. Link form actions to the grid page

When the user clicks a button, an adapter action is executed. To specify which page to display after an adapter action is executed:

1 In the Object TreeView, expand AdapterCommandGroup1 to show the CmdCancel, CmdApply, and CmdRefreshRow entries.

2 Select CmdCancel. In the Object Inspector window, type **CountryTable** in the PageName property.

3 Select CmdApply. In the Object Inspector window, type **CountryTable** in the PageName property.

Step 7. Link grid actions to the form page

To specify which page to display after an adapter action is executed by pushing a button in the grid:

1 Select View | Project Manager.

2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryTableU entry.

3 In the Object TreeView window, expand the *AdapterPageProducer* component and all its branches, to show the CmdNewRow, CmdEditRow, and CmdDeleteRow entries. These entries appear under the AdapterCommandColumn1 entry.

4 Select CmdNewRow. In the Object Inspector window, type **CountryForm** in the PageName property.

5 Select CmdEditRow. In the Object Inspector window, type **CountryForm** in the PageName property.

6 To verify that the application is working and that all buttons perform some action, run the application.

Note There will be no indication of database errors, such as an invalid type. For example, try adding a new country with an invalid value (for example, 'abc') in the Area field.

Add error reporting

To report errors to the end user, an AdapterErrorList component is used to display errors that occur while executing adapter actions that edit the country table.

Step 1. Add error support to the grid

1 In the Object TreeView for CountryTable, expand the *AdapterPageProducer* component and all its branches to show AdapterForm1.

2 Right-click on AdapterForm1 and select New Component.

3 Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Object TreeView window.

4 Move AdapterErrorList1 above AdapterGrid1 (either by dragging it or by using the upward-pointing arrow in the Object TreeView toolbar).

5 In the Object Inspector window, set the Adapter property to Adapter.

Step 2. Add error support to the form

1 In the Object TreeView for CountryForm, expand the *AdapterPageProducer* component and all its branches to show AdapterForm1.

2 Right-click on AdapterForm1 and select New Component.

3 Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Object TreeView window.

4 Move AdapterErrorList1 above AdapterGrid1 (either by dragging it or by using the upward-pointing arrow in the Object TreeView toolbar).

5 In the Object Inspector window, set the Adapter property to CountryTable.Adapter.

Step 3. Test the error-reporting mechanism

To test Grid Errors:

1 Run the application, and browse to the CountryTable page.

2 Start up another instance of your browser and browse to the CountryTable page.

3 Click the DeleteRow button on the first row in the grid.

4 Without refreshing the second browser, click the DeleteRow button on the first row in the grid.

5 An error message will be displayed above the grid.

To test Form Errors:

1 Run the application, and browse to the CountryTable page.

2 Click on the EditRow Button.

3 The CountryForm page is displayed.

4 Change the area field to 'abc', and click the Apply Button.

5 An error message will be displayed above the first field.

Run the completed application

To run the completed application:

1 Select Run | Run. You will see a form displayed. Web App Debugger executable Web applications are COM servers, and the form you see is the console window for the COM server. The first time that you run the project, it registers the COM object that can be accessed directly by Web App Debugger.

2 Select Tools | Web App Debugger.

3 Click on the default URL link to display the ServerInfo page. The ServerInfo page displays the names of all registered Web Application Debugger executables.

4 Select CountryTutorial in the drop-down list and click on the Go button.

Chapter

30

Working with XML documents

XML (Extensible Markup Language) is a markup language for describing structured data. It is similar to HTML, except that the tags describe the structure of information rather than its display characteristics. XML documents provide a simple, text-based way to store information so that it is easily searched or edited. They are often used as a standard, transportable format for data in Web applications, business-to-business communication, and so on.

XML documents provide a hierarchical view of a body of data. Tags in the XML document describe the role or meaning of each data element, as illustrated in the following document, which describes a collection of stock holdings:

```
<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM “sth.dtd”>
<StockList>
   <Stock exchange=NASDAQ>
      <name>Inprise (Borland)</name>
      <price>15.375</price>
      <symbol>INPR</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange=NYSE>
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type=preferred>25</shares>
   </Stock>
</StockList>
```

This example illustrates a number of typical elements in an XML document. The first line is a processing instruction. It provides information on how to interpret the file, but does not include any data.

The second line, which begins with the <!DOCType> tag, is a document type declaration. It names the structure of the document and references another file (sth.dtd) that describes that structure. In this case, the structure is described by a

Document Type Definition (DTD) file. Other types of files that describe the structure of an XML document include Reduced XML Data (XDR) and XML schemas (XSD).

The remaining lines are organized into a hierarchy with a single root node (the <StockList> tag). Each node in this hierarchy contains either a set of child nodes, or a text value. Some of the tags (the <Stock> and <shares> tags) include attributes, which are Name=Value pairs that provide details on how to interpret the tag.

Although it is possible to work directly with the text in an XML document, typically applications use some sort of additional tools for parsing and editing the data. W3C defines a set of standard interfaces for representing a parsed XML document called the Document Object Model (DOM). A number of vendors provide XML parsers that implement the DOM interfaces to let you interpret and edit XML documents more easily.

Delphi provides a number of additional tools for working with XML documents. These tools use a DOM parser that is provided by another vendor, and make it even easier to work with XML documents. This chapter describes those tools.

Note In addition to the tools described in this chapter, Delphi comes with tools and components for converting XML documents to data packets that integrate into the Delphi database architecture. For details on tools for integrating XML documents into database applications, see Chapter 26, "Using XML in database applications."

Using the Document Object Model

The Document Object Model (DOM) is a set of standard interfaces for representing a parsed XML document. Delphi ships with two DOM implementations (from Microsoft and from IBM). In addition, it includes a registration mechanism that lets you integrate additional DOM implementations by other vendors into the Delphi XML framework.

The XMLDOM unit includes declarations for all the DOM interfaces defined in the W3C XML DOM level 2 specification. Each DOM vendor provides an implementation for these interfaces.

- To use the Microsoft implementation, include the MSXMLDOM unit in your uses clause. Because the Microsoft implementation is COM-based, you must also register the msxml.dll library as a COM server. You can use Regsvr32.exe to register this DLL.
- To use the IBM implementation, include the IBMXMLDOM unit in your uses clause.
- To use another DOM implementation, you must create a unit that includes a function to return the top-level interface (*IDOMImplementation*). Your unit should register this function by calling the global *RegisterDOMImplementation* procedure.

Some vendors supply extensions to the standard DOM interfaces. To allow you to uses these extensions, the XMLDOM unit also defines an *IDOMNodeEx* interface. *IDOMNodeEx* is a descendant of the standard *IDOMNode* that includes the most useful of these extensions.

You can work directly with the DOM interfaces to parse and edit XML documents. Simply call the *GetDOM* function to obtain an *IDOMImplementation* interface, which you can use as a starting point.

Note For detailed descriptions of the DOM interfaces, see the declarations in the XMLDOM unit, the documentation supplied by your DOM Vendor, or the specifications provided on the W3C web site (www.w3.org).

You may find it more convenient to use Delphi's XML classes rather than working directly with the DOM interfaces. These are described below.

Working with XML components

Delphi defines a number of classes and interfaces for working with XML documents. These simplify the process of loading, editing, and saving XML documents.

Using TXMLDocument

The starting point for working with an XML document is the *TXMLDocument* component. The following steps describe how to use *TXMLDocument* to work directly with an XML document:

1. Add a *TXMLDocument* component into your form or data module. *TXMLDocument* appears on the Internet page of the Component palette.
2. Set the *DOMVendor* property to specify the DOM implementation you want the component to use for parsing and editing an XML document. The Object Inspector lists all the currently registered DOM vendors. For information on DOM implementations, see "Using the Document Object Model" on page 30-2.
3. Depending on your implementation, you may want to set the *ParseOptions* property to configure how the underlying DOM implementation parses the XML document.
4. If you are working with an existing XML document, specify the document:
   - If the XML document is stored in a file, set the *FileName* property to the name of that file.
   - You can specify the XML document as a string instead by using the *XML* property.
5. Set the *Active* property to *True*.

Once you have an active *TXMLDocument* object, you can traverse the hierarchy of its nodes, reading or setting their values. The root node of this hierarchy is available as the *DocumentElement* property.

Working with XML nodes

Once an XML document has been parsed by a DOM implementation, the data it represents is available as a hierarchy of nodes. Each node corresponds to a tagged element in the document. For example, given the following XML:

```
Component palette<?xml version="1.0" standalone='yes' ?>
<!DOCTYPE stockholdings SYSTEM “sth.dtd”>
<StockList>
   <Stock exchange=NASDAQ>
      <name>Inprise (Borland)</name>
      <price>15.375</price>
      <symbol>INPR</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange=NYSE>
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type=preferred>25</shares>
   </Stock>
</StockList>
```

TXMLDocument would generate a hierarchy of nodes as follows: The root of the hierarchy would be the *StockList* node. *StockList* would have two child nodes, which correspond to the two *Stock* tags. Each of these two child nodes would have four child nodes of its own (*name*, *price*, *symbol*, and *shares*). Those four child nodes would act as leaf nodes. The text they contain would appear as the value of each of the leaf nodes.

Note This division into nodes differs slightly from the way a DOM implementation generates nodes for an XML document. In particular, a DOM parser treats all tagged elements as internal nodes. Additional nodes (of type text node) would be created for the values of the *name*, *price*, *symbol*, and *shares* nodes. These text nodes would then appear as the children of the *name*, *price*, *symbol*, and *shares* nodes.

Each node is accessed through an *IXMLNode* interface, starting with the root node, which is the value of the XML document component’s *DocumentElement* property.

Working with a node’s value

Given an *IXMLNode* interface, you can check whether it represents an internal node or a leaf node by checking the *IsTextElement* property.

- If it represents a leaf node, you can read or set its value using the *Text* property.
- If it represents an internal node, you can access its child nodes using the *ChildNodes* property.

Thus, for example, using the XML document above, you can read the price of Inprise’s stock as follows:

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
Price := InpriseStock.ChildNodes['price'].Text;
```

Working with a node's attributes

If the node includes any attributes, you can work with them using the *Attributes* property. You can read or change an attribute value by specifying an existing attribute name. You can add new attributes by specifying a new attribute name when you set the *Attributes* property:

```
InpriseStock := XMLDocument1.DocumentElement.ChildNodes[0];
InpriseStock.ChildNodes['shares'].Attributes['type'] := 'common';
```

Adding and deleting child nodes

You can add child nodes using the *AddChild* method. *AddChild* creates new nodes that correspond to tagged elements in the XML document. Such nodes are called element nodes.

To create a new element node, specify the name that appears in the new tag and, optionally, the position where the new node should appear. For example, the following code adds a new stock listing to the document above:

```
var
  NewStock: IXMLNode;
  ValueNode: IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems'
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
  ValueNode.Text := '25';
end;
```

An overloaded version of *AddChild* lets you specify the namespace in which the tag name is defined.

You can delete child nodes using the methods of the *ChildNodes* property. *ChildNodes* is an *IXMLNodeList* interface, which manages the children of a node. You can use its *Delete* method to delete a single child node that is identified by position or by name. For example, the following code deletes the last stock listed in the document above:

```
StockList := XMLDocument1.DocumentElement;
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
```

Abstracting XML documents with the Data Binding wizard

Although it is possible to work with an XML document using only the *TXMLDocument* component and the *IXMLNode* interface it surfaces for the nodes in that document, or even to work exclusively with the DOM interfaces (avoiding even *TXMLDocument*), you can write code that is much simpler and more readable by using the XML Data Binding wizard.

The Data Binding wizard takes an XML schema or data file and generates a set of interfaces that map on top of it. For example, given XML data that looks like the following:

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

The Data Binding wizard generates the following interface (along with a class to implement it):

```
ICustomer = interface(IXMLNode)
  property id: Integer read Getid write Setid;
  property name: DOMString read Getname write Setname;
  property phone: DOMString read Getphone write Setphone;
  function Getid: Integer;
  function Getname: DOMString;
  function Getphone: DOMString;
  procedure Setid(Value: Integer);
  procedure Setname(Value: DOMString);
  procedure Setphone(Value: DOMString);
end;
```

Every child node is mapped to a property whose name matches the tag name of the child node and whose value is the interface of the child node (if the child is an internal node) or the value of the child node (for leaf nodes). Every node attribute is also mapped to a property, where the property name is the attribute name and the property value is the attribute value.

In addition to creating interfaces (and implementation classes) for each tagged element in the XML document, the wizard creates a global function for obtaining the interface to the root node. For example, if the XML above came from a document whose root node had the tag <Customers>, the Data Binding wizard would create the following global routine:

```
function GetCustomers(XMLDoc: TXMLDocument): ICustomers;
```

Using the generated interfaces simplifies your code, because they reflect the structure of the XML document more directly. For example, instead of writing code such as the following:

```
CustName := XMLDocument1.DocumentElement.ChildNodes[0].ChildNodes['name'].Value;
```

Your code would look as follows:

```
CustName := GetCustomers(XMLDocument1)[0].Name;
```

Note that the interfaces generated by the Data Binding wizard all descend from *IXMLNode*. This means you can still add and delete child nodes in the same way as when you do not use the Data Binding wizard. (See “Adding and deleting child nodes” on page 30-5.) In addition, when child nodes represent repeating elements (when all of the children of a node are of the same type), the parent node is given two methods, *Add*, and *Insert*, for adding additional repeats. These methods are simpler than using *AddChild*, because you do not need to specify the type of node to create.

Using the XML Data Binding wizard

To use the Data Binding wizard,

1 Choose File | New | Other and select the icon labeled XML Data Binding from the bottom of the New page.

2 The XML Data Binding wizard appears.

3 On the first page of the wizard

- specify the XML document or schema for which you want to generate interfaces. This can be a sample XML document, a Document Type Definition (.dtd) file, a Reduced XML Data (.xdr) file, or an XML schema (.xsd) file.

4 Click the Options button to specify the naming strategies you want the wizard to use when generating interfaces and implementation classes and the default mapping of types defined in the schema to Pascal data types.

5 Move to the second page of the wizard. This page lets you provide detailed information about every node type in the document or schema. At the left is a tree view that shows all of the node types in the document. For complex nodes (nodes that have children), the tree view can be expanded to display the child elements. When you select a node in this tree view, the right-hand side of the dialog displays information about that node and lets you specify how you want the wizard to treat that node.

- The Source Name control displays the name of the node type in the XML schema.
- The Source Type control displays the type of the node's value, as specified in the XML schema.
- The Documentation control lets you add comments to the schema describing the use or purpose of the node.
- If the wizard generates code for the selected node (that is, if it is a complex type for which the wizard generates an interface and implementation class, or if it is one of the child elements of a complex type for which the wizard generates a property on the complex type's interface), you can use the Generate Binding check box to specify whether you want the wizard to generate code for the node. If you uncheck Generate Binding, the wizard does not generate the interface or implementation class for a complex type, or does not create a property in the parent interface for a child element or attribute.
- The Binding Options section lets you influence the code that the wizard generates for the selected element. For any node, you can specify the Identifier Name (the name of the generated interface or property). In addition, for interfaces, you must indicate which one represents the root node of the document. For nodes that represent properties, you can specify the type of the property and, if the property is not an interface, whether it is a read-only property.

6 Once you have specified what code you want the wizard to generate for each node, move to the third page. This page lets you choose some global options about how the wizard generates its code and lets you preview the code that will be generated, and lets you tell the wizard how to save your choices for future use.

- To preview the code the wizard generates, select an interface in the Binding Summary list and view the resulting interface definition in the Code Preview control.
- Use the Data Binding Settings to indicate how the wizard should save your choices. You can store the settings as annotations in a schema file that is associated with the document (the schema file specified on the first page of the dialog), or you can name an independent schema file that is used only by the wizard.

7 When you click Finish, the Data Binding wizard generates a new unit that defines interfaces and implementation classes for all of the node types in your XML document. In addition, it creates a global function that takes a *TXMLDocument* object and returns the interface for the root node of the data hierarchy.

Using code that the XML Data Binding wizard generates

Once the wizard has generated a set of interfaces and implementation classes, you can use them to work with XML documents that match the structure of the document or schema you supplied to the wizard. Just as when you are using only the built-in XML components that ship with Delphi, your starting point is the *TXMLDocument* component that appears on the Internet page of the Component palette.

To work with an XML document, use the following steps:

1 Place a *TXMLDocument* component in your form or data module.

2 Bind the *TXMLDocument* to an XML document by setting the *FileName* property. (As an alternative approach, you can use a string of XML by setting the *XML* property at runtime.)

3 In your code, call the global function that the wizard created to obtain an interface for the root node of the XML document. For example, if the root element of the XML document was the tag <StockList>, by default, the wizard generates a function *GetStockListType*, which returns an *IStockListType* interface:

```
var
  StockList: IStockListType;
begin
  StockList := GetStockListType(XMLDocument1);
```

4 This interface has properties that correspond to the subnodes of the document's root element, as well as properties that correspond to that root element's attributes. You can use these to traverse the hierarchy of the XML document, modify the data in the document, and so on.

5 To save any changes you make using the interfaces generated by the wizard, call the *TXMLDocument* component's *SaveToFile* method or read its *XML* property.

Chapter 31

Using Web Services

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided.

Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as well.

Delphi's support for Web Services is designed to work using SOAP (Simple Object Access Protocol). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. It uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. For more information about SOAP, see the SOAP specification available at

```
http://www.w3.org/TR/SOAP/
```

Note Although Delphi's support for Web Services is based on SOAP and HTTP, the framework is sufficiently general that it can be expanded to use other encoding and communications protocols.

Delphi's SOAP-based technology is available on Windows and will later be implemented on Linux, so that it can form the basis of cross-platform distributed applications. There is no special client runtime software to install, as you must have when distributing applications using CORBA. Because this technology is based on HTTP messages, it has the advantage that it is widely available on a variety of machines. Support for Web Services is built on top of Delphi's cross-platform Web server application architecture.

You can use Delphi to build both servers that implement Web Services and clients that call on those services. If you use Delphi to create both the server and client applications, you can share a single unit that defines the interfaces for the Web Services. In addition, you can write Delphi clients for arbitrary servers that implement Web Services that respond to SOAP messages, and Delphi servers that publish Web Services that can be used by arbitrary clients.

When either the client or server is not written using Delphi, you can publish or import information on what interfaces are available and how to call them using a WSDL (Web Service Definition Language) document. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard can import a published WSDL document, providing you with the interface definitions and connection information you need.

Writing Servers that support Web Services

In Delphi, servers that support Web Services are built using invokable interfaces. Invokable interfaces are interfaces that are compiled to include runtime type information (RTTI). This RTTI is used when interpreting incoming method calls from clients so that they can be correctly marshaled.

In addition to the invokable interfaces, and the classes that implement them, your server requires two components: a dispatcher and an invoker. The dispatcher (*THTTPSoapDispatcher*) is an auto-dispatching component that receives incoming SOAP messages and passes them on to the invoker. The invoker (*THTTPSoapPascalInvoker*) interprets the SOAP message, identifies the invokable interface it calls, executes the call and assembles the response message.

Note *THTTPSoapDispatcher* and *THTTPSoapPascalInvoker* are designed to respond to HTTP messages containing a SOAP request. The underlying architecture is sufficiently general, however, that it can support other protocols with the substitution of different dispatcher and invoker components.

Once you register your invokable interfaces and their implementation classes, the dispatcher and invoker automatically handle any messages that identify those interfaces in the SOAP Action header of the HTTP request message.

Building a Web Service server

Use the following steps to build a server application that implements a Web Service:

1. Define the interfaces that make up your Web Service. These interface definitions must be invokable interfaces. It is a good idea to create your interface definitions in their own units, separate from the unit that contains the implementation classes. In this way, the unit that defines the interfaces can be included in both the server and client applications. In the initialization section of this unit, add code to register the interfaces. For details on writing and registering invokable interfaces, see “Defining invokable interfaces” on page 31-3.
2. If your interface uses any complex (non-scalar) types, you must make sure these can be marshalled correctly. The Web Service application can only handle these using special objects that contain runtime type information (RTTI) that describes their structure. For details on creating and registering objects to represent complex types, see “Using complex types in invokable interfaces” on page 31-5.

3 Define and implement classes that implement the invokable interfaces you defined in step 1. For each implementation class, you may also need to create a factory procedure that instantiates the class. In the initialization section of this unit, add code to register the implementation class. This process is described in “Creating and registering the implementation” on page 31-6.

4 If your application raises an exception when attempting to execute a SOAP request, the exception will be automatically encoded in a SOAP fault packet, which is returned instead of the results of the method call. If you want to convey more information than a simple error message, you can create your own exception classes that are encoded and passed to the client. This is described in “Creating custom exception classes for Web Services” on page 31-7.

5 Choose File | New | Other, and on the Web Services page, double-click the Web Service application icon. Choose the type of Web server application you want to have implement your Web Service. For information about different types of Web Server applications, see “Types of Web server applications” on page 27-6.

6 The wizard generates a new Web Service application that includes an invoker component (*THTTPSOAPPascalInvoker*) and a dispatcher component (*THTTPSoapDispatcher*). The invoker converts between SOAP messages and the methods of any interfaces you registered in step 1. The dispatcher automatically responds to incoming SOAP messages and forwards them to the invoker. You can use its *WebDispatch* property to identify the HTTP request messages to which your application responds. This involves setting the *PathInfo* property to indicate the path portion of any URL directed to your application, and the *MethodType* property to indicate the method header for request messages.

7 Choose Project | Add To Project, and add the units you created in steps 1 through 4 to your Web server application.

8 If you want your application to work with clients that are not written using Delphi, publish a WSDL document that defines your interfaces and how to call them. For details on how to generate a WSDL document that describes your Web Service application, see “Generating WSDL documents for a Web Service application” on page 31-7.

Defining invokable interfaces

To create an invokable interface, you need only compile an interface with the {$M+} compiler option. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, clients of your Web Service server can only call the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can’t be called by clients.

Delphi defines a base invokable interface, *IInvokable*, that can be used as the basis of any interface exposed to clients by a Web Service server. *IInvokable* is the same as the base interface (*IInterface*), except that it is compiled using the {$M+} compiler option so that it and all its descendants are compiled to include RTTI.

For example, the following code defines an invokable interface that contains two methods for encoding and decoding numeric values:

```
IEncodeDecode = interface(IInvokable)
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
```

Before a Web Service application can use this invokable interface, it must be registered with the invocation registry. On the server, the invocation registry entry allows the invoker component (*THTTPSOAPPascalInvoker*) to identify an implementation class to use for executing interface calls. On client applications, an invocation registry entry allows components to look up information that identifies the invokable interface and supplies information on how to call it.

In the initialization section of the unit that defines the interface, add code to register the interface with the invocation registry. To access the invocation registry, add the InvokeRegistry unit to the uses clause of your unit. The InvokeRegistry unit declares a global variable, *InvRegistry*, which maintains in memory a catalog of all registered invokable interfaces, their implementation classes, and the factories that create instances of the implementation classes.

When you are finished, the unit that defines the interface should look something like the following:

```
unit EncodeDecode;

interface
type

IEncodeDecode = interface(IInvokable)
  ['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;

implementation
uses  InvokeRegistry;

initialization
InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
```

Because the interfaces of Web Services must have a namespace to identify them among all the interfaces in all possible Web Services, when you register an interface the invocation registry automatically generates a namespace for the interface. The default namespace is built from a string that uniquely identifies the application (the *AppNamespacePrefix* variable), the interface name, and the name of the unit in which it is defined.

Tip It is a good idea to keep the unit that defines your invokable interfaces separate from the unit in which you write the classes that implement them. This unit can then be included in both the client and the server application. Because the generated namespace includes the name of the unit in which the interface is defined, sharing the same unit in both client and server applications enables them to automatically use the same namespace.

Using complex types in invokable interfaces

The invoker component (*THTTPSOAPPascalInvoker*) automatically knows how to marshal scalar types on invokable interfaces. It can also handle dynamic arrays, as long as they are registered with the remotable class registry (see below). However, you must provide additional support if you want to transmit data in more complex types such as static arrays, interfaces, records, sets, or classes. This support must take the form of a class that includes runtime type information (RTTI), which the invoker can use to convert between data in the SOAP stream and type values.

Use *TRemotable* as a base class when defining a class to represent a complex data type on an invokable interface. For example, in the case where you would ordinarily pass a record as a parameter, you would instead define a *TRemotable* descendant where every member of the record is a published property on your new class.

If the value of your new *TRemotable* descendant represents to a scalar type in a WSDL document that does not correspond to an Object Pascal scalar type, you should use *TRemotableXS* as a base class instead. *TRemotableXS* is a *TRemotable* descendant that introduces two methods for converting between your new class and its string representation. Provide these methods by overriding the *XSToNative* and *NativeToXS* methods.

In the initialization section of the unit that defines the *TRemotable* descendant, you must register this class with the remotable class registry. Access the remotable class registry by adding the InvokeRegistry unit to the uses clause. This unit declares a global variable, *RemClassRegistry*, which maintains a catalog of all registered remotable classes, and an indication of whether their values can be transmitted as strings. For example, the following line comes from the XSBuiltIns unit. It registers *TXSDateTime*, a *TRemotable* descendant that represents *TDateTime* values:

```
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNameSpace, 'dateTime', True);
```

The first parameter is the name of the *TRemotable* descendant. The second is a uniform resource identifier (URI) that uniquely identifies the namespace of the new class. If you supply an empty string, the registry can generate a URI for you. The third parameter is the name of the data type your class represents. If you supply an empty string, the registry simply uses the class name. The last parameter indicates whether the value of class instances can be transmitted as a string (whether you implemented the *XSToNative* and *NativeToXS* methods).

Tip It is a good idea to implement and register *TRemotable* descendants in a separate unit from the rest of your server application, including from the units that declare and register invokable interfaces. In this way, you can use the unit that defines your type in both the client and server, and you can use the type for more than one interface.

If you are using dynamic arrays for parameters, you do not need to create a remotable class to represent them, but you do have to register them with the remotable class registry. Thus, for example, if your interface uses a type such as the following:

```
type
  TDateTimeArray = array of TXSDateTime;
```

You must add the following registration to the initialization section of the unit where you declare this dynamic array:

```
RemClassRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNameSpace, ‘DTarray’, False);
```

The parameters are the same as those used by *RegisterXSClass*, except for the first which takes a pointer to the type information of the dynamic array rather than a class reference.

Creating and registering the implementation

The simplest way to write an implementation for an invokable interface is to create a class that descends from *TInvokableClass*. Add the class declaration, including the invokable interfaces you are supporting, and then type *Ctrl+Shift+C* to invoke class completion. The interface members appear in your class declaration, and empty methods appear in the implementation section of the unit.

For example, the declaration for an implementation class implement the interface declared in "Defining invokable interfaces" above might look like the following:

```
TEncodeDecode = class(TInvokableClass, IEncodeDecode)
protected
  function EncodeValue(Value: Integer): Double; stdcall;
  function DecodeValue(Value: Double): Integer; stdcall;
end;
```

In the implementation section of the unit that declares this class, fill in the *EncodeValue* and *DecodeValue* methods.

Once you have created an implementation class, you must register this class with the invocation registry. The invocation registry uses this to identify the class that implements a registered interface and to make it available to the invoker component when the invoker needs to call the interface. To register the implementation class, add a call the *RegisterInvokableClass* method of the global *InvRegistry* variable to the initialization section of your implementation unit:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode);
```

You can also create implementation classes that do not descend from *TInvokableClass*. In this case, however, you must provide a factory procedure that the invocation registry can call to create instances of your class.

The factory procedure must be of type *TCreateInstanceProc*. It returns an instance of your implementation class. If the procedure creates a new instance, the object should free itself when the reference count on its interface drops to zero, as the invocation registry does not explicitly free object instances. As an alternative, the factory procedure can return a reference to a global instance that is shared by all callers. The following code illustrates this latter approach:

```
procedure CreateEncodeDecode(out obj: TObject);
begin
  if FEncodeDecode = nil then
  begin
    FEncodeDecode := TEncodeDecode.Create;
    {save a reference to the interface so that the global instance doesn't free itself }
    FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
  end;
  obj := FEncodeDecode; { return global instance }
end;
```

When using a factory procedure, supply the factory procedure as a second parameter to the *RegisterInvokableClass* method:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
```

Creating custom exception classes for Web Services

When your Web Service application raises an exception in the course of trying to execute a SOAP request, it automatically encodes information about that exception in a SOAP fault packet, which it returns instead of the results of the method call. The client application then raises the exception.

By default, the client application merely raises a generic exception (*Exception*) with the error message in the SOAP fault packet. However, you can transmit additional exception information by using an exception class that descends from *ERemotableException*. The values of any published properties you add to your exception class are included in the SOAP fault packet so that the client can raise an equivalent exception.

To use an *ERemotableException* descendant, you must register it with the remotable class registry. Thus, in the unit that defines your *ERemotableException* descendant, you must add the InvokeRegistry unit to the uses clause and add a call to the *RegisterXSClass* method of the global *RemClassRegistry* variable.

If the client uses the same unit that defines and registers your *ERemotableException* descendant, then when it receives the SOAP fault packet, it automatically raises an instance of the appropriate exception class, with all properties set to the values in the SOAP fault packet.

Generating WSDL documents for a Web Service application

If you include the same units that define and register your invokable interfaces, the classes that represent complex type information, and your remotable exceptions in a Delphi client application, it can generate calls to use your Web Service. All you need to do is supply the URL where you install your Web Service application.

However, you may want to make your Web Service available to a wider range of clients. For example, you may have clients that are not written in Delphi. If you are deploying several versions of your server application, you may not want to use a single hard-coded URL for the server, but rather let the client look up the server location dynamically. For these cases, you may want to publish a WSDL document that describes the types and interfaces in your Web Service, with information on how to call them.

To publish a WSDL document that describes your Web Service, simply add a *TWSDLHTMLPublish* component to your Web Module. *TWSDLHTMLPublish* is an auto-dispatching component, which means it automatically responds to incoming messages that request a list of WSDL documents for your Web Service. Use the *WebDispatch* property to specify the path information of the URL clients must use to access the list of WSDL documents. The Web browser can then request the list of WSDL documents by specifying an URL that is made up of the location of the server application followed by the path in the WebDispatch property. This URL looks something like the following:

```
http://www.myco.com/MyService.dll/WSDL
```

Tip If you want a physical WSDL file instead, you can display the WSDL document in your Web browser and then save it to generate a WSDL document file.

It is not necessary to publish the WSDL document from the same application that implements your Web Service. To create an application that simply publishes the WSDL document, omit the units that contain the implementation objects, and only include the units that define and register invokable interfaces, remotable classes that represent complex types, and any remotable exceptions.

By default, when you publish a WSDL document, it indicates that the services are available at the same URL as the one where you published the WSDL document (but with a different path). If you are deploying multiple versions of your Web Service application, or if you are publishing the WSDL document from a different application than the one that implements the Web Service, you will need to change the WSDL document so that includes updated information on where to locate the Web Service.

To change the URL, use the WSDL administrator. The first step is to enable the administrator. You do this by setting the *AdminEnabled* property of the *TWSDLHTMLPublish* component to *True*. Then, when you use your browser to display the list of WSDL documents, it includes a button to administer them as well. Use the WSDL administrator to specify the locations (URLs) where you have deployed your Web Service application.

Writing clients for Web Services

Delphi provides client-side support for calling Web Services that use a SOAP-based binding. These Web Services can be supplied by a server written in Delphi, or by any other server that defines its Web Service in a WSDL document.

If the server is not written in Delphi, you can first import the WSDL document that describes the server. This process is described below. If the server was written using Delphi, you do not need to use a WSDL document: you can simply add any units that define the invokable interfaces you want to use to your project, as well as any units that define remotable classes that represent complex types and that define remotable exceptions that the Web Service application can raise.

Note For information about creating the unit that defines an invokable interface in a Delphi Web Service server, see “Defining invokable interfaces” on page 31-3. For information about creating a unit that defines a remotable class for a complex type, see “Using complex types in invokable interfaces” on page 31-5. For information about creating a unit that defines a remotable exception, see “Creating custom exception classes for Web Services” on page 31-7.

Importing WSDL documents

Before you can use a Web Service that was not written using Delphi, you must import a WSDL document (or XML schema file) that defines the service. The Web Services importer creates a unit that defines and registers the interfaces and types you need to use.

To use the Web Services importer, choose File | New | Other, and on the WebServices page double-click the icon labelled Web Services importer. In the dialog that appears, specify the file name of a WSDL document (or XML schema file) or provide the URL where that document is published. When you click Generate, the importer creates new units that define and register invokable interfaces for the operations defined in the document, and that define and register remotable classes for the types that the document defines.

If the WSDL document or XML schema file uses identifiers that are also Object Pascal keywords, the importer automatically adjusts their names so that the generated code can compile. When complex types are declared inline, the importer adds code to define and register the corresponding remotable class in the same unit as the invokable interface that uses them. Otherwise, types are defined and registered in a separate unit.

Calling invokable interfaces

To call an invokable interface, your client application must include any units that define the invokable interfaces and any remotable classes that implement complex types. If the server is written in Delphi, these should be the same units that the server application uses to define and register these interfaces and classes. It is best to use the same unit, because when you register an invokable interface or remotable class, it is given a uniform resource identifier (URI) that uniquely identifies it. That URI is derived from the name of the interface (or class) and the name of the unit in which it is defined. If the client and server do not register the interface (or class) using the same URI, they can’t communicate. If you do not use the same unit, the code that registers the interface and implementation class must explicitly specify a namespace URI to ensure that client and server use the same namespace.

If the server is not written in Delphi, or if you do not want to use the same unit in the client that you used in the server, these units can be created by the Web Services importer.

Once the client application has the declaration of an invokable interface, create an instance of *THTTPRio* for the desired interface:

```
X := THTTPRio.Create(nil);
```

Next, provide the *THTTPRio* object with the information it needs to identify the server interface and locate the server. There are two ways to supply this information:

- If the server is written in Delphi, the identification of the interface on the server is handled automatically, based on the URI that is generated for it when the interface is registered. You need only set the *URL* property to indicate the location of the server. The path portion of this URL should match the path of the dispatcher component in the server’s Web Module:

  ```
  X.URL := 'http://www.myco.com/MyService.dll/SOAP/';
  ```

- If the server is not written in Delphi, *THTTPRio* must look up the URI for the interface, the information that must be included in the Soap Action header, and the location of the server from a WSDL document. You tell it how to do this using the *WSDLLocation*, *Service*, and *Port* properties:

  ```
  X.WSDLLocation := 'Cryptography.wsdl';
  X.Service := 'Cryptography';
  X.Port := 'SoapEncodeDecode';
  ```

You can then use the **as** operator to cast the instance of *THTTPRio* to the invokable interface. When you do this, it creates a vtable for the associated interface dynamically in memory, enabling you to make interface calls:

```
InterfaceVariable := X as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
```

THTTPRio relies on the invocation registry to obtain information about the invokable interface. If the client application does not have an invocation registry, or if the invokable interface is not registered, *THTTPRio* can’t build its in-memory vtable.

Chapter

32

Working with sockets

This chapter describes the socket components that let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as User Datagram Protocol (UDP), Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

Implementing services

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

Understanding service protocols

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the clients that use it. You can copy the API of a standard third party server (such as Apache), or you can design and publish your own API.

Services and ports

Most standard services are associated, by convention, with specific port numbers. We will discuss port numbers in greater detail later. For now, consider the port number a numeric code for the service.

If you are implementing a standard service, Linux socket objects provide methods for you to look up the port number for the service. If you are providing a new service, you can specify the associated port number in the /etc/services file. See your Linux documentation for more information on the services file.

Types of socket connections

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections.
- Listening connections.
- Server connections.

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

Client connections

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket to which it wishes to connect. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

Listening connections

Server sockets do not locate clients. Instead, they form passive “half connections” that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

Server connections

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

Describing sockets

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies

- The system on which it is running.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host and the port number.

Before you can make a socket connection, you must fully describe the sockets that form its endpoints. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

Describing the host

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.ASite.com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. You can learn the host name associated with any IP address (if one already exists) by executing the following command from a command prompt:

```
nslookup IPADDRESS
```

where *IPADDRESS* is the IP address you're interested in. If your local IP address doesn't have a host name and you decide you want one, contact your network administrator.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

Using ports

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

Earlier, we described port numbers as numeric codes for the services implemented by network applications. This is actually just a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

Using socket components

The Internet palette page includes three socket components that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. These are:

- *TcpServer*
- *TcpClient*
- *UdpSocket*

Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the socket server calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the socket objects.

Getting information about the connection

After completing the connection to a client or server socket, you can use the client or server socket object associated with your socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the local client or server socket, or use the *RemoteHost* and *RemotePort* properties to determine the address and port number used by the remote client or server socket. Use the *GetSocketAddr* method to build a valid socket address based on the host name and port number. You can use the *LookupPort* method to look up the port number. Use the *LookupProtocol* method to look up the protocol number. Use the *LookupHostName* method to look up the host name based on the host machine's IP address.

To view network traffic in and out of the socket, use the *BytesSent* and *BytesReceived* properties.

Using client sockets

Add a *TcpClient* or *UdpSocket* component to your form or data module to turn your application into a TCP/IP or UDP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Each client socket component uses a single client socket object to represent the client endpoint in a connection.

Specifying the desired server

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the *RemoteHost* property to specify the remote host server by either its host name or IP address.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can use the *RemotePort* property to specify the server port number directly or indirectly by naming the target service.

Forming the connection

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the *Open* method. If you want your application to form the connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

Getting information about the connection

After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the *Handle* property to obtain a handle to the socket connection to use when making socket calls.

Closing the connection

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the *Close* method. The connection may also be closed from the server end. If that is the case, you will receive notification in an *OnDisconnect* event.

Using server sockets

Add a server socket component (*TcpServer* or *UdpSocket*) to your form or data module to turn your application into an IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

Each server socket component uses a single server socket object to represent the server endpoint in a listening connection. It also uses a server client socket object for the server endpoint of each active connection to a client socket that the server accepts.

Specifying the port

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the *LocalPort* property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.

Listening for client requests

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the *Open* method. If you want your application to form the listening connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

Connecting to clients

A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an *OnAccept* event.

Closing server connections

When you want to shut down the listening connection, call the *Close* method or set the *Active* property to *False*. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When TCP clients shut down their individual connections to your server socket, you are informed by an *OnDisconnect* event.

Responding to socket events

When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf* , *SendStream*, or *Sendln* methods in your program after the socket has been opened. You can read from the socket using the similarly-named methods *ReceiveBuf* and *Receiveln.* The *OnSend* and *OnReceive* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive two events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, you must use the *SendBuf* and *ReceiveBuf* methods to respond to these client events or server events.

Error events

Client and server sockets generate *OnError* events when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about

- What socket object received the error notification.
- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

Client events

When a client socket opens a connection, the following events occur:

- The socket is set up and initialized for event notification.
- An *OnCreateHandle* event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection.
- The connection request is accepted by the server and completed by the client socket.
- When the connection is established, the *OnConnect* notification event occurs.

Server events

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

Events when listening

Just before the listening connection is formed, the *OnListening* event occurs. You can use its *Handle* property to make changes to the socket before it is opened for listing. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListening* event handler.

Events with client connections

When a server socket accepts a client connection request, the following events occur:

- An *OnAccept* event occurs, passing in the new *TTcpClient* object to the event handler. This is the first point when you can use the properties of *TTcpClient* to obtain information about the server endpoint of the connection to a client.
- If *BlockMode* is *bmThreadBlocking* an *OnGetThread* event occurs. If you want to provide your own customized descendant of *TServerSocketThread*, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerSocketThread*. If you want to perform any initialization of the thread, or make any socket API calls before the thread starts reading or writing over the connection, you should use the *OnGetThread* event handler for these tasks as well.
- The client completes the connection and an *OnAccept* event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

Reading and writing over socket connections

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection. You can also form blocking connections, where your application waits for the reading or writing to be completed before executing the next line of code.

Non-blocking connections

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection for client or server sockets, set the *BlockMode* property to *bmNonBlocking.*

When the connection is non-blocking, reading and writing events inform your socket when the socket on the other end of the connection tries to read or write information.

Reading and writing events

Non-blocking sockets generate reading and writing events when it needs to read or write over the connection. You can respond to these notifications in an *OnReceive* or *OnSend* event handler.

The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the *ReceiveBuf* or *Receiveln* method. To write to the socket connection, use the *SendBuf*, *SendStream*, or *Sendln* method.

Blocking connections

When the connection is blocking your socket must initiate reading or writing over the connection rather than waiting passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client or server sockets, set the *BlockMode* property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *BlockMode* property to *bmBlocking* or *bmThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *BlockMode* is *bmThreadBlocking*. When the *BlockMode* is *bmBlocking*, program execution is blocked until a new connection is established.

Part

IV

Developing COM-based applications

The chapters in “Developing COM-based applications” present concepts necessary for building COM-based applications, including Automation controllers, Automation servers, ActiveX controls, and COM+ applications.

Note Support for COM clients is available in all editions of Delphi. However, to create servers, you need the Professional or Enterprise edition.

Chapter

33

Overview of COM technologies

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

Note COM components such as those on the ActiveX, COM+, and Servers tabs of the Component palette are not available for use in CLX applications. This technology is for use on Windows only and is not cross-platform.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM). For more information on clients, servers, and interfaces see, “Parts of a COM application,” on page 33-3.

This chapter provides a conceptual overview of the underlying technology on which Automation and ActiveX controls are built. Later chapters provide details on creating Automation objects and ActiveX controls in Delphi.

COM as a specification and implementation

COM is both a specification and an implementation. The COM specification defines how objects are created and how they communicate with each other. According to this specification, COM objects can be written in different languages, run in different process spaces and on different platforms. As long as the objects adhere to the written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is built into the Win32 subsystem, which provides a number of core services that support the written specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions designed for the purpose of creating and managing COM objects.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly (such as Active Documents). You can find these wrappers defined in the ComObj unit and the API definitions in the AxCtrls unit.

Note Delphi's interfaces and language follow the COM specification. Delphi implements objects conforming to the COM spec using a set of classes called the Delphi ActiveX framework (DAX). These classes are found in the AxCtrls, OleCtrls, and OleServer units. In addition, the Pascal interface to the COM API is in ActiveX.pas and ComSvcs.pas.

COM extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories. For details on COM extensions, see "COM extensions" on page 33-10.

In addition, when working in a large, distributed environment, you can create transactional COM objects. Prior to Windows 2000, these objects were not architecturally part of COM, but rather ran in the Microsoft Transaction Server (MTS) environment. With the advent of Windows 2000, this support is integrated into COM+. Transactional objects are described in detail in Chapter 39, "Creating MTS or COM+ objects."

Delphi provides wizards to easily implement applications that incorporate the above technologies in the Delphi environment. For details, see "Implementing COM objects with wizards" on page 33-18.

Parts of a COM application

When implementing a COM application, you supply the following:

COM Interface The way in which an object exposes its services externally to clients. A COM object provides an interface for each set of related methods and properties. Note that COM properties are not identical to properties on VCL objects. COM properties always use read and write access methods.

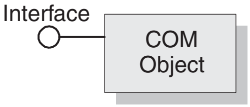
COM server A module, either an EXE, DLL, or OCX, that contains the code for a COM object. Object implementations reside in servers. A COM object implements one or more interfaces.

COM client The code that calls the interfaces to get the requested services from the server. Clients know what they want to get from the server (through the interface); clients do not know the internals of how the server provides the services. Delphi eases the process in creating a client by letting you install COM servers (such as a Word document or PowerPoint slide) as components on the Component Palette. This allows you to connect to the server and hook its events through the Object Inspector.

COM interfaces

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients. The standard way to depict a COM interface is shown in Figure 33.1:

Figure 33.1 A COM interface



For example, every COM object implements the basic interface, *IUnknown,* which tells the client what interfaces are available on the COM object.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to convey to the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces are immutable; that is, they do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.
- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as *IMalloc* or *IPersist*.

- Interfaces are guaranteed to have a unique identification, called a **Globally Unique Identifier (GUID)**, which is a 128-bit randomly generated number. Interface GUIDs are called **Interface Identifiers (IIDs)**. This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer either explicitly or implicitly.
- Interfaces are not objects themselves; they provide a way to access an object. Therefore, clients do not access data directly; clients access data through an interface pointer. Windows 2000 adds an additional layer of indirection known as an interceptor through which it provides COM+ features such as just-in-time activation and object pooling.
- Interfaces are always inherited from the fundamental interface, *IUnknown*.
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection. For more information see , "In-process, out-of-process, and remote servers," on page 33-6.

The fundamental COM interface, IUnknown

All COM objects must support the fundamental interface, called *IUnknown*, a **typedef** to the base interface type *IInterface*. *IUnknown* contains the following routines:

QueryInterface	Provides pointers to other interfaces that the object supports.
AddRef and Release	Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

Clients obtain pointers to other interfaces through the *IUnknown* method, *QueryInterface. QueryInterface* knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any method of the interface.

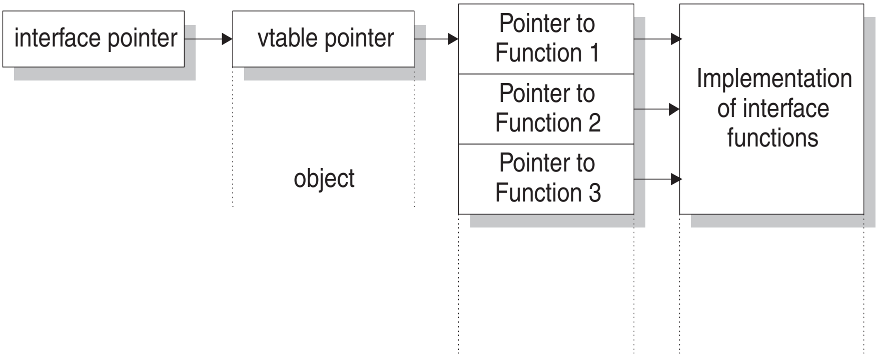
Objects track their own lifetime through the *IUnknown* methods, *AddRef* and *Release*, which are simple reference counting methods. As long as an object's reference count is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object(s).

COM interface pointers

An interface pointer is a 32-bit pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a **vtable**. Vtables are similar to the mechanism used to support virtual functions in Object Pascal. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Object Pascal classes.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's interface pointer, then, is a pointer *to the pointer* to the vtable, as shown in the following diagram.

Figure 33.2 Interface vtable



In Windows 2000 and subsequent versions of Windows, when an object is running under COM+, an added level of indirection is provided between the interface pointer and the vtable pointer. The interface pointer available to the client points at an interceptor, which in turn points at the vtable. This allows COM+ to provide such services as just-in-time activation, whereby the server can be deactivated and reactivated dynamically in a way that is opaque to the client. To achieve this, COM+ guarantees that the interceptor behaves as if it were an ordinary vtable pointer.

COM servers

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods.

Clients do not know *how* a COM object performs its service; the object's implementation remains encapsulated. An object makes its services available through its interfaces as described previously.

In addition, clients do not need to know *where* a COM object resides. COM provides transparent access regardless of the object's location.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server instantiate an object instance for the client. This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

As a minimum, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).

- Implement a class factory object, which manufactures another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

Note Delphi wizards automate the creation of COM objects and servers as described in "Implementing COM objects with wizards" on page 33-18.

CoClasses and class factories

A COM object is an instance of a **CoClass**, which is a class that implements one or more COM interfaces. The COM object provides the services as defined by its interfaces.

CoClasses are instantiated by a special type of object called a *class factory*. Whenever an object's services are requested by a client, a class factory creates an object instance for that particular client. Typically, if another client requests the object's services, the class factory creates another object instance to service the second client. (Clients can also bind to running COM objects that register themselves to support it.)

A CoClass must have a class factory and a class identifier (CLSID) so that it can be instantiated externally, that is, from another module. Using these unique identifiers for CoClasses means that they can be updated whenever new interfaces are implemented in their class. A new interface can modify or add methods without affecting older versions, which is a common problem when using DLLs.

Delphi wizards take care of assigning class identifiers and of implementing and instantiating class factories.

In-process, out-of-process, and remote servers

With COM, a client does not need to know where an object resides, it simply makes a call to an object's interface. COM performs the necessary steps to make the call. These steps differ depending on whether the object resides in the same process as the client, in a different process on the client machine, or in a different machine across the network. The different types of servers are known as:

In-process server A library (DLL) running in the *same process space* as the client, for example, an ActiveX control embedded in a Web page viewed under Internet Explorer or Netscape. Here, the ActiveX control is downloaded to the client machine and invoked within the same process as the Web browser.

The client communicates with the in-process server using direct calls to the COM interface.

Out-of-process server (or local server) Another application (EXE) running in a *different process space* but on the *same machine* as the client. For example, an Excel spreadsheet embedded in a Word document are two separate applications running on the same machine.

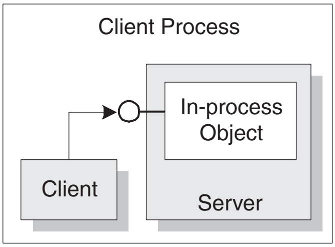
The local server uses COM to communicate with the client.

Remote server A DLL or another application running on a *different machine* from that of the client. For example, a Delphi database application is connected to an application server on another machine in the network.

The remote server uses distributed COM (DCOM) to access interfaces and communicate with the application server.

As shown in Figure 33.3, for in-process servers, pointers to the object interfaces are in the same process space as the client, so COM makes direct calls into the object implementation.

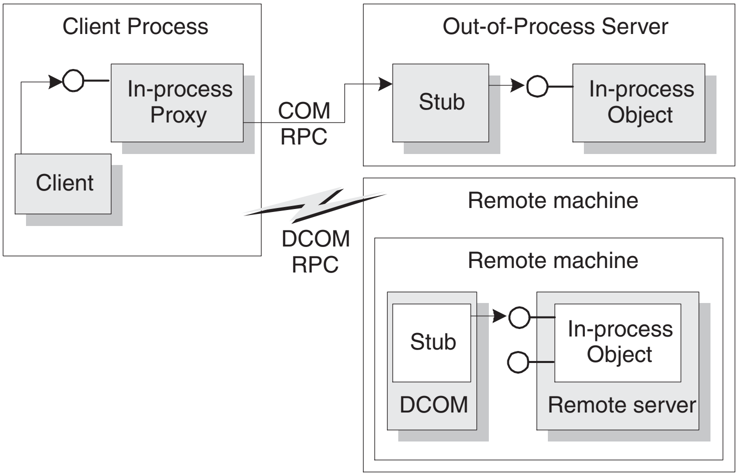
Figure 33.3 In-process server



Note This is not always true under COM+. When a client makes a call to an object in a different context, COM+ intercepts the call so that it behaves like a call to an out-of-process server (see below), even if the server is in-process. See Chapter 39, “Creating MTS or COM+ objects” for more information working with COM+.

As shown in Figure 33.4, when the process is either in a different process or in a different machine altogether, COM uses a proxy to initiate remote procedure calls. The **proxy** resides in the same process as the client, so from the client’s perspective, all interface calls look alike. The proxy intercepts the client’s call and forwards it to where the real object is running. The mechanism that enables the client to access objects in a different process space, or even different machine, as if they were in their own process, is called **marshaling.**

Figure 33.4 Out-of-process and remote servers



The difference between out-of-process and remote servers is the type of interprocess communication used. The proxy uses COM to communicate with an out-of-process server, it uses distributed COM (DCOM) to communicate with a remote machine. DCOM transparently transfers a local object request to the remote object running on a different machine.

Note For remote procedure calls, DCOM uses the RPC protocol provided by Open Group's Distributed Computing Environment (DCE). For distributed security, DCOM uses the NT LAN Manager (NTLM) security protocol. For directory services, DCOM uses the Domain Name System (DNS).

The marshaling mechanism

Marshaling is the mechanism that allows a client to make interface function calls to remote objects in another process or on a different machine. Marshaling

- Takes an interface pointer in the server's process and makes a proxy pointer available to code in the client process.
- Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space.

For any interface call, the client pushes arguments onto a stack and makes a function call through the interface pointer. If the call to the object is not in-process, the call gets passed to the proxy. The proxy packs the arguments into a marshaling packet and transmits the structure to the remote object. The object's stub unpacks the packet, pushes the arguments onto the stack, and calls the object's implementation. In essence, the object recreates the client's call in its own address space.

What type of marshaling occurs depends on what the COM object implements. Objects can use a standard marshaling mechanism provided by the *IDispatch* interface. This is a generic marshaling mechanism that enables communication through a system-standard remote procedure call (RPC). For details on the *IDispatch* interface, see "Automation interfaces" on page 36-12. Even if the object does not implement *IDispatch*, if it limits itself to automation-compatible types and has a registered type library, COM automatically provides marshaling support.

Applications that do not limit themselves to automation-compatible types or register a type library must provide their own marshaling. Marshaling is provided either through an implementation of the *IMarshal* interface, or by using a separately generated proxy/stub DLL. Delphi does not support the automatic generation of proxy/stub DLLs.

Aggregation

Sometimes, a server object makes use of another COM object to perform some of its functions. For example, an inventory management object might make use of a separate invoicing object to handle customer invoices. If the inventory management object wants to present the invoice interface to clients, however, there is a problem: Although a client that has the inventory interface can call *QueryInterface* to obtain the invoice interface, when the invoice object was created it did not know about the inventory management object and can't return an inventory interface in response to a call to *QueryInterface*. A client that has the invoice interface can't get back to the inventory interface.

To avoid this problem, some COM objects support **aggregation**. When the inventory management object creates an instance of the invoice object, it passes it a copy of its own *IUnknown* interface. The invoice object can then use that *IUnknown* interface to handle any *QueryInterface* calls that request an interface, such as the inventory interface, that it does not support. When this happens, the two objects together are called an aggregate. The invoice object is called the inner, or contained object of the aggregate, and the inventory object is called the outer object.

Note In order to act as the outer object of an aggregate, a COM object must create the inner object using the Windows API *CoCreateInstance* or *CoCreateInstanceEx*, passing its *IUnknown* pointer as a parameter that the inner object can use for *QueryInterface* calls.

In order to create an object that can act as the inner object of an aggregate, it must descend from *TContainedObject*. When the object is created, the *IUnknown* interface of the outer object is passed to the constructor so that it can be used by the *QueryInterface* method on calls that the inner object can't handle.

COM clients

Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the *IDispatch* interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations about the client using its objects. Similarly, clients don't need to know how (or even where) an object provides the services; they simply rely on server objects to provide the services they advertise through their interfaces.

There are two types of COM clients, controllers and containers. Controllers launch the server and interact with it through its interface. They request services from the COM object or drive it as a separate process. Containers host visual controls or objects that appear in the container’s user interface. They use predefined interfaces to negotiate display issues with server objects. It is impossible to have a container relationship over DCOM; for example, visual controls that appear in the container's user interface must be located locally. This is because the controls are expected to paint themselves, which requires that they have access to local GDI resources.

Delphi makes it easier for you to develop COM clients by letting you import a type library or ActiveX control into a component wrapper so that server objects look like other VCL components. For details on this process, see Chapter 35, “Creating COM clients”.

COM extensions

COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes.

The following lists some of the services COM extensions currently provide. Subsequent sections describe these services in greater detail.

Automation servers Automation refers to the ability of an application to control the objects in another application programmatically. Automation servers are the objects that can be controlled by other executables at runtime.

ActiveX controls ActiveX controls are specialized in-process servers, typically intended for embedding in a client application. The controls offer both design and runtime behaviors as well as events.

Active Server Pages Active Server Pages are scripts that generate HTML pages. The scripting language includes constructs for creating and running Automation objects. That is, the Active Server Page acts as an Automation controller.

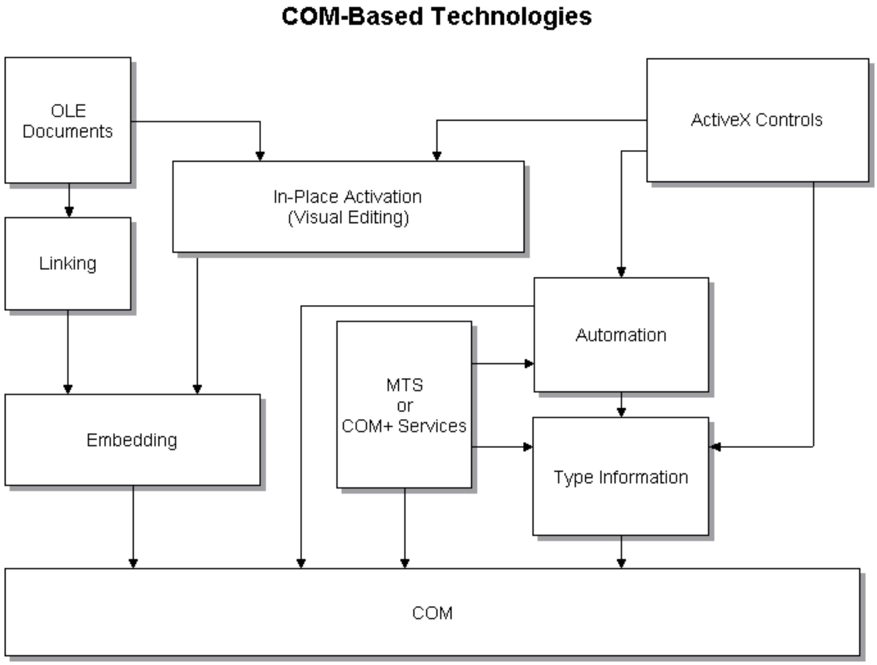
Active Documents Objects that support linking and embedding, drag-and-drop, visual editing, and in-place activation. Word documents and Excel spreadsheets are examples of Active Documents.

Transactional objects Objects that include additional support for responding to large numbers of clients. This includes features such as just-in-time activation, transactions, resource pooling, and security services. These features were originally handled by MTS but have been built into COM with the advent of COM+.

Type libraries A collection of static data structures, often saved as a resource, that provides detailed type information about an object and its interfaces. Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available.

The following diagram illustrates the relationship of the COM extensions and how they are built upon COM:

Figure 33.5 COM-based technologies



COM objects can be visual or non-visual. Some must run in the same process space as their clients; others can run in different processes or remote machines, as long as the objects provide marshaling support. Table 33.1 summarizes the types of COM objects that you can create, whether they are visual, process spaces they can run in, how they provide marshaling, and whether they require a type library.

Table 33.1 COM object requirements

Object	Visual Object?	Process space	Communication	Type library
Active Document	Usually	In-process, or out-of-process	OLE Verbs	No
Automation	Occasionally	In-process, out-of-process, or remote	Automatically marshaled using the *IDispatch* interface (for out-of process and remote servers)	Required for automatic marshaling
ActiveX Control	Usually	In-process	Automatically marshaled using the *IDispatch* interface	Required
MTS or COM+	Occasionally	In-process for MTS, any for COM+	Automatically marshaled via a type library	Required

Table 33.1 COM object requirements (continued)

Object	Visual Object?	Process space	Communication	Type library
In-process custom interface object	Optionally	In-process	No marshaling required for in-process servers	Recommended
Other custom interface object	Optionally	In-process, out-of-process, or remote	Automatically marshaled via a type library; otherwise, manually marshaled using custom interfaces	Recommended

Automation servers

Automation refers to the ability of an application to control the objects in another application programmatically, like a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller.

Automation can be used on in-process, local, and remote servers.

Automation is characterized by two key points:

- The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide information about its interfaces, the interface methods, and those methods' arguments. Typically, this information is available in a type library. The Automation server can also generate type information dynamically when queried via its *IDispatch* interface (see following).
- Automation objects make their methods accessible so that other applications can use them. For this, they implement the *IDispatch* interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space because the Automation *IDispatch* interface automates the marshaling process. Automation does, however, restrict the types that you can use.

For a list of types that are valid for type libraries in general, and Automation interfaces in particular, see "Valid types" on page 34-11.

For information on writing an Automation server, see Chapter 36, "Creating simple COM servers."

Active Server Pages

The Active Server Page (ASP) technology lets you write simple scripts, called Active Server Pages, that can be launched by clients via a Web server. Unlike ActiveX controls, which run on the client, Active Server Pages run on the server, and return a resulting HTML page to clients.

Active Server Pages are written in Jscript or VB script. The script runs every time the server loads the Web page. That script can then launch an embedded Automation server (or Enterprise Java Bean). For example, you can write an Automation server, such as one to create a bitmap or connect to a database, and this server accesses data that gets updated every time a client loads the Web page.

Active Server Pages rely on the Microsoft Internet Information Server (IIS) environment to serve your Web pages.

Delphi wizards let you create an Active Server Object, which is an Automation object specifically designed to work with an Active Server Page. For more information about creating and using these types of objects, see Chapter 37, "Creating an Active Server Page."

ActiveX controls

ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications that need to be downloaded by a client before they are used.

ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as prefabricated OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display issues with their host containers.

ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

One use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX is a standard that targets interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

Delphi wizards allow you to easily create ActiveX controls. For more information about creating and using these types of objects, see Chapter 38, "Creating an ActiveX control."

Active Documents

Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, *TOleContainer*, to support linking and embedding of existing Active Documents.

You can also use *TOleContainer* as a basis for an Active Document container. To create objects for Active Document servers, use the COM object wizard and add the appropriate interfaces, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

Note While the specification for Active Documents has built-in support for marshaling in cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine such as window handles, menu handles, and so on.

Transactional objects

Delphi uses the term “transactional objects” to refer to objects that take advantage of the transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) (for versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later). These objects are designed to work in a large, distributed environment.

The transaction services provide robustness so that activities are always completed or rolled back (the server never partially completes an activity). The security services allow you to expose different levels of support to different classes of clients. The resource management allows an object to handle more clients by pooling resources or keeping objects active only when they are in use. To enable the system to provide these services, the object must implement the *IObjectControl* interface. To access the services, transactional objects use an interface called *IObjectContext*, which is created on their behalf by MTS or COM+.

Under MTS, the server object must be built into a library (DLL), which is then installed in the MTS runtime environment. That is, the server object is an in-process server that runs in the MTS runtime process space. Under COM+, this restriction does not apply because all COM calls are routed through an interceptor. To clients, the difference between MTS and COM+ is transparent.

MTS or COM+ servers group transactional objects that run in the same process space. Under MTS, this group is called an MTS package, while under COM+ it is called a COM+ application. A single machine can be running several different MTS packages (or COM+ applications), where each one is running in a separate process space.

To clients, the transactional object may appear like any other COM server object. The client need never know about transactions, security, or just-in-time activation unless it is initiating a transaction itself.

Both MTS and COM+ provide a separate tool for administering transactional objects. This tool lets you configure objects into packages or COM+ applications, view the packages or COM+ applications installed on a computer, view or change the attributes of the included objects, monitor and manage transactions, make objects available to clients, and so on. Under MTS, this tool is the MTS Explorer. Under COM+ it is the COM+ Component Manager.

Type libraries

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available. All of Delphi's wizards generate a type library automatically, although the COM object wizard makes this optional. You can view or edit this type information by using the Type Library Editor as described in Chapter 34, "Working with type libraries."

This section describes what a type library contains, how it is created, when it is used, and how it is accessed. For developers wanting to share interfaces across languages, the section ends with suggestions on using type library tools.

The content of type libraries

Type libraries contain *type information*, which indicates which interfaces exist in which COM objects, and the types and numbers of arguments to the interface methods. These descriptions include the unique identifiers for the CoClasses (CLSIDs) and the interfaces (IIDs), so that they can be properly accessed, as well as the dispatch identifiers (dispIDs) for Automation interface methods and properties.

Type libraries can also contain the following information:

- Descriptions of custom type information associated with custom interfaces
- Routines that are exported by the Automation or ActiveX server, but that are not interface methods
- Information about enumeration, record (structures), unions, alias, and module data types
- References to type descriptions from other type libraries

Creating type libraries

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then running that script through a compiler. However, Delphi automatically generates a type library when you create a COM object (including ActiveX controls, Automation objects, remote data modules, and so on) using any of the wizards on the ActiveX or Multitier page of the new items dialog. (You can opt not to create a type library when using the COM object wizard.) You can also create a type library by choosing from the main menu, File | New | Other, select the ActiveX tab, and choose Type Library.

You can view the type library using Delphi's Type Library editor. You can easily edit your type library using the Type Library editor and Delphi automatically updates the corresponding .tlb file (binary type library file) when the type library is saved. For any changes to Interfaces and CoClasses that were created using a wizard, the Type Library editor also updates your implementation files. For more information on using the Type Library editor to write interfaces and CoClasses, see Chapter 34, "Working with type libraries."

When to use type libraries

It is important to create a type library for each set of objects that is exposed to external users, for example,

- ActiveX controls require a type library, which must be included as a resource in the DLL that contains the ActiveX controls.
- Exposed objects that support vtable binding of custom interfaces must be described in a type library because vtable references are bound at compile time. Clients import information about the interfaces from the type library and use that information to compile. For more information about vtable and compile time binding, see "Automation interfaces" on page 36-12.
- Applications that implement Automation servers should provide a type library so that clients can early bind to it.
- Objects instantiated from classes that support the *IProvideClassInfo* interface, such as all descendants of the VCL *TTypedComObject* class, must have a type library.
- Type libraries are not required, but are useful for identifying the objects used with OLE drag-and-drop.

When defining interfaces for internal use only (within an application) you do not need to create a type library.

Accessing type libraries

The binary type library is normally a part of a resource file (.res) or a stand-alone file with a .tlb file-name extension. When included in a resource file, the type library can be bound into a server (.dll, .ocx, or .exe).

Once a type library has been created, object browsers, compilers, and similar tools can access type libraries through special type interfaces:

Interface	Description
ITypeLib	Provides methods for accessing a library of type descriptions.
ITypeLib2	Augments *ITypeLib* to include support for documentation strings, custom data, and statistics about the type library.
ITypeInfo	Provides descriptions of individual objects contained in a type library. For example, a browser uses this interface to extract information about objects from the type library.
ITypeInfo2	Augments *ITypeInfo* to access additional type library information, including methods for accessing custom data elements.
ITypeComp	Provides a fast way to access information that compilers need when binding to an interface.

Delphi can import and use type libraries from other applications by choosing Project | Import Type Library. Most of the VCL classes used for COM applications support the essential interfaces that are used to store and retrieve type information from type libraries and from running instances of an object. The VCL class *TTypedComObject* supports interfaces that provide type information, and is used as a foundation for the ActiveX object framework.

Benefits of using type libraries

Even if your application does not require a type library, you can consider the following benefits of using one:

- Type checking can be performed at compile time.
- You can use early binding with Automation, and controllers that do not support vtables or dual interfaces can encode dispIDs at compile time, improving runtime performance.
- Type browsers can scan the library, so clients can see the characteristics of your objects.
- The *RegisterTypeLib* function can be used to register your exposed objects in the registration database.
- The *UnRegisterTypeLib* function can be used to completely uninstall an application’s type library from the system registry.
- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.

Using type library tools

The tools for working with type libraries are listed below.

- The TLIBIMP (Type Library Import) tool, which takes existing type libraries and creates Delphi Interface files (_TLB.pas files), is incorporated into the Type Library editor. TLIBIMP provides additional configuration options not available inside the Type Library editor.
- TRegSvr is a tool for registering and unregistering servers and type libraries, which comes with Delphi. The source to TRegSvr is available as an example in the Demos directory.
- The Microsoft IDL compiler (MIDL) compiles IDL scripts to create a type library.
- RegSvr32.exe is a tool for registering and unregistering servers and type libraries, which is a standard Windows utility.
- OLEView is a type library browser tool, found on Microsoft's Web site.

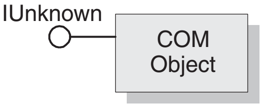
Implementing COM objects with wizards

Delphi makes it easier to write COM server applications by providing wizards that handle many of the details involved. Delphi provides separate wizards to create the following:

- A simple COM object
- An Automation object
- An Active Server Object (for embedding in an Active Server page)
- An ActiveX control
- An ActiveX Form
- A transactional object
- A Property page
- A Type library
- An ActiveX library

The wizards handle many of the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. As shown in Figure 33.6, with a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.

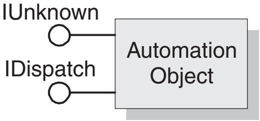
Figure 33.6 Simple COM object interface



The COM object wizard also provides an implementation for *IDispatch* if you specify that you are creating an object that supports an *IDispatch* descendant.

As shown inFigure 33.7, for Automation and Active Server objects, the wizard implements *IUnknown* and *IDispatch*, which provides automatic marshaling.

Figure 33.7 Automation object interface



As shown in Figure 33.8, for ActiveX control objects and ActiveX forms, the wizard implements all the required ActiveX control interfaces, from *IUnknown*, *IDispatch*, *IOleObject*, *IOleControl*, and so on. For a complete list of interfaces, see the reference page for *TActiveXControl* object.

Figure 33.8 ActiveX object interface

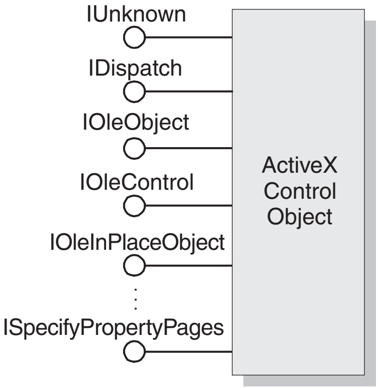


Table 33.2 lists the various wizards and the interfaces they implement:

Table 33.2 Delphi wizards for implementing COM, Automation, and ActiveX objects

Wizard	Implemented interfaces	What the wizard does
COM server	*IUnknown* (and *IDispatch* if you select a default interface that descends from *IDispatch*)	Exports routines that handle server registration, class registration, loading and unloading the server, and object instantiation. Creates and manages class factories for objects implemented on the server. Provides registry entries for the object that specify the selected threading model. Declares the methods that implement a selected interface, providing skeletal implementations for you to complete. Provides a type library, if requested. Allows you to select an arbitrary interface that is registered in the type library and implement it. If you do this, you must use a type library.

Table 33.2 Delphi wizards for implementing COM, Automation, and ActiveX objects (continued)

Wizard	Implemented interfaces	What the wizard does
Automation server	*IUnknown*, *IDispatch*	Performs the tasks of a COM server wizard (described above), plus: Implements the interface that you specify, either dual or dispatch. Provides server-side support for generating events, if requested. Provides a type library automatically.
Active Server Object	*IUnknown*, *IDispatch*, *(IASPObject)*	Performs the tasks of an Automation object wizard (described above) and optionally generates an .ASP page which can be loaded into a Web browser. It leaves you in the Type Library editor so that you can modify the object's properties and methods if needed. Surfaces the ASP intrinsics as properties so that you can easily obtain information about the ASP application and the HTTP messages that launched it.
ActiveX Control	*IUnknown*, *IDispatch*, *IPersistStreamInit*, *IOleInPlaceActiveObject*, *IPersistStorage*, *IViewObject*, *IOleObject*, *IViewObject2*, *IOleControl*, *IPerPropertyBrowsing*, *IOleInPlaceObject*, *ISpecifyPropertyPages*	Performs the tasks of the Automation server wizard (described above), plus: Generates a CoClass that corresponds to the VCL control on which the ActiveX control is based and which implements all the ActiveX interfaces. Leaves you in the source code editor so that you can modify the implementation class.
ActiveForm	Same interfaces as ActiveX Control	Performs the tasks of the ActiveX control wizard, plus: Creates a *TActiveForm* descendant that takes the place of the pre-existing VCL class in the ActiveX control wizard. This new class lets you design the Active Form the same way you design a form in a Windows application.
Transactional object	*IUnknown*, *IDispatch*, *IObjectControl*	Adds a new unit to the current project containing the MTS or COM+ object definition. It inserts proprietary GUIDs into the type library so that Delphi can install the object properly, and leaves you in the Type Library editor so that you can define the interface that the object exposes to clients. You must install the object separately after it is built.
Property Page	*IUnknown*, *IPropertyPage*	Creates a new property page that you can design in the Forms designer.
COM+ Event object	None, by default	Creates a COM+ event object that you can define using the Type Library editor. Unlike the other object wizards, the COM+ Event object wizard does not create an implementation unit because event objects have no implementation (it is provided by client sinks).

Table 33.2 Delphi wizards for implementing COM, Automation, and ActiveX objects (continued)

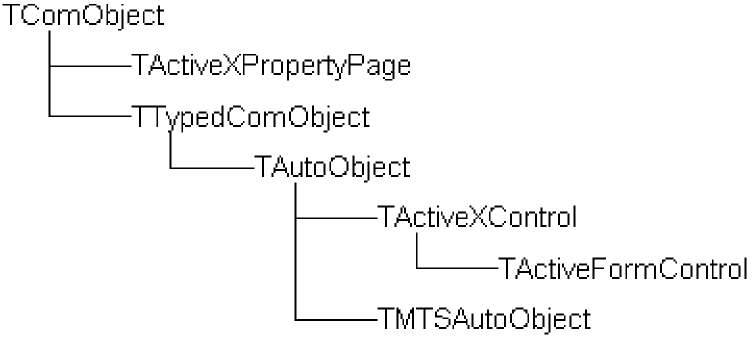
Wizard	Implemented interfaces	What the wizard does
Type Library	None, by default	Creates a new type library and associates it with the active project.
ActiveX library	None, by default	Creates a new ActiveX or Com server DLL and exposes the necessary export functions.

You can add additional COM objects or reimplement an existing implementation. To add a new object, it is easiest to use the wizard a second time. This is because the wizard sets up an association between the type library and an implementation class, so that changes you make in the type library editor are automatically applied to your implementation object.

Code generated by wizards

Delphi's wizards generate classes that are derived from the Delphi ActiveX framework (DAX). Despite its name, the Delphi ActiveX framework supports all types of COM objects, not just ActiveX controls. The classes in this framework provide the underlying implementation of the standard COM interfaces for the objects you create using a wizard. Figure 33.9 illustrates the objects in the Delphi ActiveX framework:

Figure 33.9 Delphi ActiveX framework



Each wizard generates an implementation unit that implements your COM server object. The COM server object (the implementation object) descends from one of the classes in DAX:

Table 33.3 DAX Base classes for generated implementation classes

Wizard	Base class from DAX	Inherited support
COM server	TTypedCOMObject	Support for *IUnknown* and *ISupportErrorInfo* interfaces. Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces. Support for reading type library information.

Table 33.3 DAX Base classes for generated implementation classes (continued)

Wizard	Base class from DAX	Inherited support
Automation server Active Server Object	TAutoObject	Everything provided by *TTypedCOMObject*, plus: Support for the *IDispatch* interface. Auto-marshaling support.
ActiveX Control	TActiveXControl	Everything provided by *TAutoObject*, plus: Support for embedding in a container. Support for in-place activation. Support for properties and property pages. The ability to delegate to an associated windowed control that it creates.
ActiveForm	TActiveFormControl	Everything provided by *TAutoObject*, except that it works with a descendant of *TActiveForm* rather than another windowed control class.
MTS object	TMTSAutoObject	Everything provided by *TAutoObject*, plus: Support for the *IObjectControl* interface.
Property Page	TPropertyPage (uses TActiveXPropertyPage internally)	Support for *IUnknown* and *ISupportErrorInfo* interfaces. Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces. Support for the *IPropertyPage* interface.

Corresponding to the classes in Figure 33.9 is a hierarchy of class factory objects that handle the creation of these COM objects. The wizard adds code to the initialization section of your implementation unit that instantiates the appropriate class factory for your implementation class.

The wizards also generate a type library and its associated unit, which has a name of the form Project1_TLB. The Project1_TLB unit includes the definitions your application needs to use the type definitions and interfaces defined in the type library. For more information on the contents of this file, see “Code generated when you import type library information” on page 35-5.

You can modify the interface generated by the wizard using the type library editor. When you do this, the implementation class is automatically updated to reflect those changes. You need only fill in the bodies of the generated methods to complete the implementation.

Chapter

34

Working with type libraries

This chapter describes how to create and edit type libraries using Delphi's Type Library editor. Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by a COM object. They provide a way to identify what types of objects and interfaces are available on a server. For a detailed overview on why and when to use type libraries, see "Type libraries" on page 33-15.

A type library can contain any and all of the following:

- Information about custom data types such as aliases, enumerations, structures, and unions.
- Descriptions of one or more COM elements, such as an interface, dispinterface, or CoClass. Each of these descriptions is commonly referred to as *type information.*
- Descriptions of constants and methods defined in external units.
- References to type descriptions from other type libraries.

By including a type library with your COM application or ActiveX library, you make information about the objects in your application available to other applications and programming tools through COM's type library tools and interfaces.

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then run that script through a compiler. The Type Library editor automates some of this process, easing the burden of creating and modifying your own type libraries.

When you create a COM server of any type (ActiveX control, Automation object, remote data module, and so on) using Delphi's wizards, the wizard automatically generates a type library for you (although in the case of the COM object wizard, this is optional). Most of the work you do in customizing the generated object starts with the type library, because that is where you define the properties and methods it exposes to clients: you change the interface of the CoClass generated by the wizard, using the Type Library editor. The Type Library editor automatically updates the implementation unit for your object, so that all you need do is fill in the bodies of the generated methods.

You can also use the Delphi Type Library Editor in the development of Common Object Request Broker Architecture (CORBA) applications. With traditional CORBA tools, you must define object interfaces separately from your application, using the CORBA Interface Definition Language (IDL). You then run a utility that generates stub-and-skeleton code from that definition. However, Delphi generates the stub, skeleton, and IDL for you automatically. You can easily edit your interface using the Type Library editor and Delphi automatically updates the appropriate source files.

Type Library editor

The Type Library editor enables developers to examine and create type information for COM objects. Using the Type Library editor can greatly simplify the task of developing COM objects by centralizing the tasks of defining interfaces, CoClasses, and types, obtaining GUIDs for new interfaces, associating interfaces with CoClasses, updating implementation units, and so on.

Note The Type Library editor is also used to define CORBA interfaces in projects that use the CORBA Object or CORBA Data Module wizard.

The Type Library editor outputs two types of file that represent the contents of the type library:

Table 34.1 Type Library editor files

File	Description
.TLB file	The binary type library file. By default, you do not need to use this file, because the type library is automatically compiled into the application as a resource. However, you can use this file to explicitly compile the type library into another project or to deploy the type library separately from the .exe or .ocx. For more information, see “Opening an existing type library” on page 34-19 and “Deploying type libraries” on page 34-27. **Note:** When using the Type Library editor for CORBA interfaces, the Type Library editor does not create the .tlb file.
_TLB unit	This unit interprets the contents of the type library for use by your application. It contains all the declarations your application needs to use the elements defined in the type library. Although you can open this file in the code editor, you should never edit it—it is maintained by the Type Library editor, so any changes you make will be overwritten by the Type Library editor. For more details on the contents of this file, see “Code generated when you import type library information” on page 35-5. **Note:** When using the Type Library editor for CORBA interfaces, this unit defines the stub and skeleton objects required by the CORBA application.

Parts of the Type Library editor

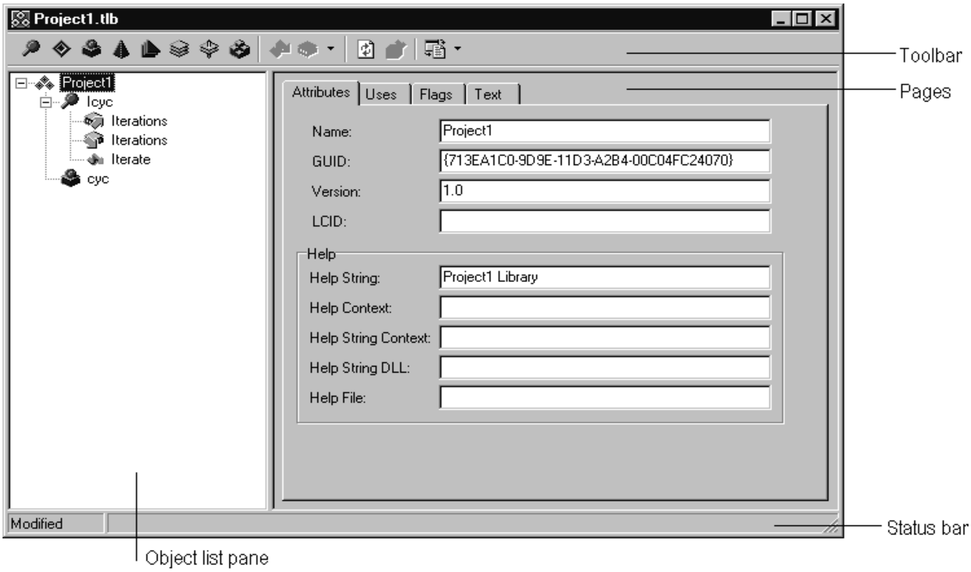
The main elements of the Type Library editor are described in Table 34.2:

Table 34.2 Type Library editor parts

Part	Description
Toolbar	Includes buttons to add new types, CoClasses, interfaces, and interface members to your type library. The toolbar also includes buttons for refreshing your implementation unit, registering the type library, and saving an IDL file with the information in your type library.
Object list pane	Displays all the existing elements in the type library. When you click on an item in the object list pane, it displays pages valid for that object.
Status bar	Displays syntax errors if you try to add invalid types to your type library.
Pages	Display information about the selected object. Which pages appear here depends on the type of object selected.

These parts are illustrated in Figure 34.1, which shows the Type Library editor displaying type information for a COM object named cyc.

Figure 34.1 Type Library editor



Name: Project1
GUID: {713EA1C0-9D9E-11D3-A2B4-00C04FC24070}
Version: 1.0
LCID:

Help
Help String: Project1 Library
Help Context:
Help String Context:
Help String DLL:
Help File:

Toolbar

The Type Library editor's toolbar located at the top of the Type Library Editor, contains buttons that you click to add new objects into your type library.

The first group of buttons let you add elements to the type library. When you click a toolbar button, the icon for that element appears in the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you select, different pages of information appear to the right.

The following table lists the elements you can add to your type library:

Icon	Meaning
	An interface description.
	A dispinterface description. (not used for CORBA interface definitions)
	A CoClass.
	An enumeration.
	An alias.
	A record.
	A union.
	A module.

When you select one of the elements listed above in the object list pane, the second group of buttons displays members that are valid for that element. For example, when you select Interface, the Method and Property icons in the second box become enabled because you can add methods and properties to your interface definition. When you select Enum, the second group of buttons changes to display the Const member, which is the only valid member for Enum type information.

The following table lists the members that can be added to elements in the object list pane:

Icon	Meaning
	A method of the interface, dispinterface, or an entry point in a module.
	A property on an interface or dispinterface.
	A write-only property. (available from the drop-down list on the property button)
	A read-write property. (available from the drop-down list on the property button)
	A read-only property. (available from the drop-down list on the property button)
	A field in a record or union.
	A constant in an enum or a module.

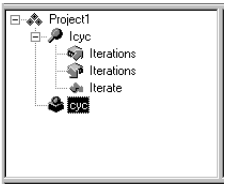
In the third box, you can choose to refresh, register, or export your type library (save it as an IDL file), as described in “Saving and registering type library information” on page 34-24.

Object list pane

The Object list pane displays all the elements of the current type library in a tree view. The root of the tree represents the type library itself, and appears as the following icon:

Descending from the type library node are the elements in the type library:

Figure 34.2 Object list pane



When you select any of these elements (including the type library itself), the pages of type information to the right change to reflect only the relevant information for that element. You can use these pages to edit the definition and properties of the selected element.

You can manipulate the elements in the object list pane by right clicking to get the object list pane context menu. This menu includes commands that let you use the Windows clipboard to move or copy existing elements as well as commands to add new elements or customize the appearance of the Type Library editor.

Status bar

When editing or saving a type library, syntax, translation errors, and warnings are listed in the Status bar pane.

For example, if you specify a type that the Type Library editor does not support, you will get a syntax error. For a complete list of types supported by the Type Library editor, see “Valid types” on page 34-11.

Pages of type information

When you select an element in the object list pane, pages of type information appear in the Type Library editor that are valid for the selected element. Which pages appear depends on the element selected in the object list panel, as follows:

Table 34.3 Type library pages

Type Info element	Page of type information	Contents of page
Type library	Attributes	Name, version, and GUID for the type library, as well as information linking the type library to help.
	Uses	List of other type libraries that contain definitions on which this one depends.
	Flags	Flags that determine how other applications can use the type library.
	Text	All definitions and declarations defining the type library itself (see discussion below).
Interface	Attributes	Name, version, and GUID for the interface, the name of the interface from which it descends, and information linking the interface to help.
	Flags	Flags that indicate whether the interface is hidden, dual, Automation-compatible, and/or extensible.
	Text	The definitions and declarations for the Interface (see discussion below).
Dispinterface	Attributes	Name, version, and GUID for the interface, and information linking it to help.
	Flags	Flags that indicate whether the Dispinterface is hidden, dual, and/or extensible.
	Text	The definitions and declarations for the Dispinterface. (see discussion below).
CoClass	Attributes	Name, version, and GUID for the CoClass, and information linking it to help.
	Implements	A List of interfaces that the CoClass implements, as well as their attributes.
	COM+	The attributes of transactional objects, such as the transaction model, call synchronization, just-in-time activation, object pooling, and so on. Also includes the attributes of COM+ event objects.
	Flags	Flags that indicate various attributes of the CoClass, including how clients can create and use instances, whether it is visible to users in a browser, whether it is an ActiveX control, and whether it can be aggregated (act as part of a composite).
	Text	The definitions and declarations for the CoClass (see discussion below).
Enumeration	Attributes	Name, version, and GUID for the enumeration, and information linking it to help.
	Text	The definitions and declarations for the enumerated type (see discussion below).
Alias	Attributes	Name, version, and GUID for the enumeration, the type the alias represents, and information linking it to help.

Table 34.3 Type library pages (continued)

Type Info element	Page of type information	Contents of page
	Text	The definitions and declarations for the alias (see discussion below).
Record	Attributes	Name, version, and GUID for the record, and information linking it to help.
	Text	The definitions and declarations for the record (see discussion below).
Union	Attributes	Name, version, and GUID for the union, and information linking it to help.
	Text	The definitions and declarations for the union (see discussion below).
Module	Attributes	Name, version, GUID, and associated DLL for the module, and information linking it to help.
	Text	The definitions and declarations for the module (see discussion below).
Method	Attributes	Name, dispatch ID or DLL entry point, and information linking it to help.
	Parameters	Method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the method, whether this is a default method for the interface, and whether it is replaceable.
	Text	The definitions and declarations for the method (see discussion below).
Property	Attributes	Name, dispatch ID, type of property access method (getter vs. setter), and information linking it to help.
	Parameters	Property access method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the property, whether this is a default for the interface, whether the property is replaceable, bindable, and so on.
	Text	The definitions and declarations for the property access method (see discussion below).
Const	Attributes	Name, value, type (for module consts), and information linking it to help.
	Flags	Flags to indicate how clients can view and use the constant, whether this represents a default value, whether the constant is bindable, and so on.
	Text	The definitions and declarations for the constant (see discussion below).
Field	Attributes	Name, type, and information linking it to help.
	Flags	Flags to indicate how clients can view and use the field, whether this represents a default value, whether the field is bindable, and so on.
	Text	The definitions and declarations for the field (see discussion below).

Note For more detailed information about the various options you can set on type information pages, see the online Help for the Type Library editor.

You can use each of the pages of type information to view or edit the values it displays. Most of the pages organize the information into a set of controls so that you can type in values or select them from a list without requiring that you know the syntax of the corresponding declarations. This can prevent many small mistakes such as typographic errors when specifying values from a limited set. However, you may find it faster to type in the declarations directly. To do this, use the Text page.

All type library elements have a text page that displays the syntax for the element. This syntax appears in an IDL subset of Microsoft Interface Definition Language, or Object Pascal. Any changes you make in other pages of the element are reflected on the text page. If you add code directly in the text page, changes are reflected in the other pages of the Type Library editor.

The Type Library editor generates syntax errors if you add identifiers that are currently not supported by the editor; the editor currently supports only those identifiers that relate to type library support (not RPC support or constructs used by the Microsoft IDL compiler for C++ code generation or marshaling support).

Type library elements

The Type Library interface can seem overwhelmingly complicated at first. This is because it represents information about a great number of elements, each of which has its own characteristics. However, many of these characteristics are common to all elements. For example, every element (including the type library itself) has the following:

- A Name, which is used to describe the element and which is used when referring to the element in code.
- A GUID (globally unique identifier), which is a globally unique 128-bit value that COM uses to identify the element. This should always be supplied for the type library itself and for CoClasses and interfaces. It is optional otherwise.
- A Version number, which distinguishes between multiple versions of the element. This is always optional, but should be provided for CoClasses and interfaces, because some tools can’t use them without a version number.
- Information linking the element to a Help topic. These include a Help String, and Help Context or Help String Context value. The Help Context is used for a traditional Windows Help system where the type library has a stand-alone Help file. The Help String Context is used when help is supplied by a separate DLL instead. The Help Context or Help String Context refers to a Help file or DLL that is specified on the type library’s Attributes page. This is always optional.

Interfaces

An interface describes the methods (and any properties expressed as ‘get’ and ‘set’ functions) for an object that must be accessed through a virtual function table (VTable). If an interface is flagged as dual, a dispinterface is also implied and can be accessed through OLE automation. By default, the type library flags all interfaces you add as dual.

Interfaces can be assigned members: methods and properties. These appear in the object list pane as children of the interface node. Properties for interfaces are represented by the 'get' and 'set' methods used to read and write the property's underlying data. They are represented in the tree view using special icons that indicate their purpose.

Note When a property is specified as Write By Reference, it means it is passed as a pointer rather than by value. Some applications, such a Visual Basic, use Write By Reference, if it is present, to optimize performance. To pass the property only by reference rather than by value, use the property type *By Reference Only*. To pass the property by reference as well as by value, select Read | Write | Write By Ref. To invoke this menu, go to the toolbar and select the arrow next to the property icon.

Once you add the properties or methods using the toolbar button or the object list pane context menu, you desribe their syntax and attributes by selecting the property or method and using the pages of type information.

The Attributes page lets you give the property or method a name and dispatch ID (so that it can be called using IDispatch). For properties, you also assign a type. The function signature is created using the Parameters page, where you can add, remove, and rearrange parameters, set their type and any modifiers, and specify function return types.

Note Members of interfaces that need to raise exceptions should return an HRESULT and specify a return value parameter (PARAM_RETVAL) for the actual return value. Declare these methods using the **safecall** calling convention.

Note that when you assign properties and methods to an interface, they are implicitly assigned to its associated CoClass. This is why the Type Library editor does not let you add properties and methods directly to a CoClass.

Dispinterfaces

Interfaces are more commonly used than dispinterfaces to describe the properties and methods of an object. Dispinterfaces are only accessible through dynamic binding, while interfaces can have static binding through a vtable.

You can add methods and properties to dispinterfaces in the same way you add them to interfaces. However, when you create a property for a dispinterface, you can't specify a function kind or parameter types.

CoClasses

A CoClass describes a unique COM object that implements one or more interfaces. When defining a CoClass, you must specify which implemented interface is the default for the object, and optionally, which dispinterface is the default source for events. Note that you do not add properties or methods to a CoClass in the Type Library editor. Properties and methods are exposed to clients by interfaces, which are associated with the CoClass using the Implements page.

Type definitions

Enumerations, aliases, records, and unions all declare types that can then be used elsewhere in the type library.

Enums consist of a list of constants, each of which must be numeric. Numeric input is usually an integer in decimal or hexadecimal format. The base value is zero by default. You can add constants to your enumeration by selecting the enumeration in the object list pane and clicking the Const button on the toolbar or selecting New | Const command from the object list pane context menu.

Note It is strongly recommended that you provide help strings for your enumerations to make their meaning clearer. The following is a sample entry of an enumeration type for a mouse button and includes a help string for each enumeration element.

```
mbLeft = 0 [helpstring 'mbLeft'];

mbRight = 1 [helpstring 'mbRight'];

mbMiddle = 3 [helpstring 'mbMiddle'];
```

An alias creates an alias (type definition) for a type. You can use the alias to define types that you want to use in other type info such as records or unions. Associate the alias with the underlying type definition by setting the Type attribute on the Attributes page.

A record consists of a list of structure members or fields. A union is a record with only a variant part. Like a record, a union consists of a list of structure members or fields. However, unlike the members of records, each member of a union occupies the same physical address, so that only one logical value can be stored.

Add the fields to a record or union by selecting it in the object list pane and clicking the field button in the toolbar or right clicking and choosing field from the object list pane context menu. Each field has a name and a type, which you assign by selecting the field and assigning values using the Attributes page. Records and unions can be defined with an optional tag.

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Modules

A module defines a group of functions, typically a set of DLL entry points. You define a module by

- Specifying a DLL that it represents on the attributes page.
- Adding methods and constants using the toolbar or the object list pane context menu. For each method or constant, you must then define its attributes by selecting the it in the object list pane and setting the values on the Attributes page.

For module methods, you must assign a name and DLL entry-point using the attributes page. Declare the function's parameters and return type using the parameters page.

For module constants, use the Attributes page to specify a name, type, and value.

Note The Type Library editor does not generate any declarations or implementation related to a module. The specified DLL must be created as a separate project.

Using the Type Library editor

Using the type library editor, you can create new type libraries or edit existing ones. Typically, an application developer uses a wizard to create the objects that are exposed in the type libarary, letting Delphi generate the type library automatically. Then, the automatically-generated type library is opened in the Type Library editor so that the interfaces can be defined (or modified), type definitions added, and so on.

However, even if you are not using a wizard to define the objects, you can use the Type Library editor to define a new type library. In this case, you must create any implementation classes yourself, because the Type Library editor does not generate code for CoClasses that were not associated with a type library by a wizard.

The editor supports a subset of valid types in a type library as described below.

The final topics in this section describe how to:

- Create a new type library
- Open an existing type library
- Add an interface to the type library
- Modify an interface
- Add properties and methods to the type library
- Add a CoClass to the type library
- Add an interface to a CoClass
- Add an enumeration to the type library
- Add an alias to the type library
- Add a record or union to the type library
- Add a module to the type library
- Save and register type library information

Valid types

In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Object Pascal. Specify the language you want to use in the Environment options dialog.

The following types are valid in a type library for COM development. The Automation compatible column specifies whether the type can be used by an interface that has its Automation or Dispinterface flag checked.

These are the types that COM can marshal via the type library automatically.

Table 34.4 Valid types

Pascal type	IDL type	variant type	Automation compatible	Description
Smallint	short	VT_I2	Yes	2-byte signed integer
Integer	long	VT_I4	Yes	4-byte signed integer
Single	single	VT_R4	Yes	4-byte real
Double	double	VT_R8	Yes	8-byte real
Currency	CURRENCY	VT_CY	Yes	currency
TDateTime	DATE	VT_DATE	Yes	date
WideString	BSTR	VT_BSTR	Yes	binary string
IDispatch	IDispatch	VT_DISPATCH	Yes	pointer to IDispatch interface
SCODE	SCODE	VT_ERROR	Yes	Ole Error Code
WordBool	VARIANT_BOOL	VT_BOOL	Yes	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	Yes	Ole Variant
IUnknown	IUnknown	VT_UNKNOWN	Yes	pointer to IUnknown interface
Shortint	byte	VT_I1	No	1 byte signed integer
Byte	unsigned char	VT_UI1	Yes	1 byte unsigned integer
Word	unsigned short	VT_UI2	Yes*	2 byte unsigned integer
LongWord	unsigned long	VT_UI4	Yes*	4 byte unsigned integer
Int64	__int64	VT_I8	No	8 byte signed integer
Largeuint	uint64	VT_UI8	No	8 byte unsigned integer
SYSINT	int	VT_INT	Yes*	system dependent integer (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	Yes*	system dependent unsigned integer
HResult	HRESULT	VT_HRESULT	No	32 bit error code
Pointer		VT_PTR -> VT_VOID	No	untyped pointer
SafeArray	SAFEARRAY	VT_SAFEARRAY	No	OLE Safe Array
PChar	LPSTR	VT_LPSTR	No	pointer to Char
PWideChar	LPWSTR	VT_LPWSTR	No	pointer to WideChar

* Word, LongWord, SYSINT, and SYSUINT are Automation-compatible in most applications, but in older applications they may not be.

Note The Byte (VT_UI1) is Automation-compatible, but is not allowed in a Variant or OleVariant since many Automation servers do not handle this value correctly.

Besides these IDL types, any interfaces and types defined in the library or defined in referenced libraries can be used in a type library definition.

The Type Library editor stores type information in the generated type library (.TLB) file in binary form.

If a parameter type is specified as a Pointer type, the Type Library editor usually translates that type into a variable parameter. When the type library is saved, the variable parameter’s associated ElemDesc’s IDL flags are marked IDL_FIN or IDL_FOUT.

Often, ElemDesc IDL flags are not marked by IDL_FIN or IDL_FOUT when the type is preceded with a Pointer. Or, in the case of dispinterfaces, IDL flags are not typically used. In these cases, you may see a comment next to the variable identifier such as {IDL_None} or {IDL_In}. These comments are used when saving a type library to correctly mark the IDL flags.

SafeArrays

COM requires that arrays be passed via a special data type known as a SafeArray. You can create and destroy SafeArrays by calling special COM functions to do so, and all elements within a SafeArray must be valid automation-compatible types. The Delphi compiler has built-in knowledge of COM SafeArrays and automatically calls the COM API to create, copy, and destroy SafeArrays.

In the Type Library editor, a *SafeArray* must specify the type of its elements. For example, the following line from the text page declares a method with a parameter that is a *SafeArray* with an element type of Integer:

```
procedure HighLightLines(Lines: SafeArray of Integer);
```

Note Although you must specify the element type when declaring a *SafeArray* type in the Type Library editor, the declaration in the generated _TLB unit does not indicate the element type.

Using Object Pascal or IDL syntax

The Text page of the Type Library editor displays your type information in one of two ways:

- Using an extension of Object Pascal syntax.
- Using the Microsoft IDL.

Note When working on a CORBA object, you use neither of these on the text page. Instead, you must use the CORBA IDL.

You can select which language you want to use by changing the setting in the Environment Options dialog. Choose Tools | Environment Options, and specify either Pascal or IDL as the Language on the Type Library page of the dialog.

Note The choice of Object Pascal or IDL syntax also affects the choices available on the parameters attributes page.

Like Object Pascal applications in general, identifiers in type libraries are case insensitive. They can be up to 255 characters long, and must begin with a letter or an underscore (_).

Attribute specifications

Object Pascal has been extended to allow type libraries to include attribute specifications. Attribute specifications appear enclosed in square brackets and separated by commas. Each attribute specification consists of an attribute name followed (if appropriate) by a value.

The following table lists the attribute names and their corresponding values.

Table 34.5 Attribute syntax

Attribute name	Example	Applies to
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo
bindable	[bindable]	members except CoClass members
control	[control]	type library, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	anything
default	[default]	CoClass members
defaultbind	[defaultbind]	members except CoClass members
defaultcollection	[defaultcollection]	members except CoClass members
defaultvtbl	[defaultvtbl]	CoClass members
dispid	[dispid]	members except CoClass members
displaybind	[displaybind]	members except CoClass members
dllname	[dllname 'Helper.dll']	module typeinfo
dual	[dual]	interface typeinfo
helpfile	[helpfile 'c:\help\myhelp.hlp']	type library
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	type library
helpcontext	[helpcontext 2005]	anything except CoClass members and parameters
helpstring	[helpstring 'payroll interface']	anything except CoClass members and parameters
helpstringcontext	[helpstringcontext $17]	anything except CoClass members and parameters
hidden	[hidden]	anything except parameters
immediatebind	[immediatebind]	members except CoClass members
lcid	[lcid $324]	type library
licensed	[licensed]	type library, CoClass typeinfo
nonbrowsable	[nonbrowsable]	members except CoClass members
nonextensible	[nonextensible]	interface typeinfo
oleautomation	[oleautomation]	interface typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	members except CoClass members
propput	[propput]	members except CoClass members

Table 34.5 Attribute syntax (continued)

Attribute name	Example	Applies to
propputref	[propputref]	members except CoClass members
public	[public]	alias typeinfo
readonly	[readonly]	members except CoClass members
replaceable	[replaceable]	anything except CoClass members and parameters
requestedit	[requestedit]	members except CoClass members
restricted	[restricted]	anything except parameters
source	[source]	all members
uidefault	[uidefault]	members except CoClass members
usesgetlasterror	[usesgetlasterror]	members except CoClass members
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	type library, typeinfo (required)
vararg	[vararg]	members except CoClass members
version	[version 1.1]	type library, typeinfo

Interface syntax

The Object Pascal syntax for declaring interface type information has the form

```
interfacename = interface[(baseinterface)] [attributes]
functionlist
[propertymethodlist]
end;
```

For example, the following text declares an interface with two methods and one property:

```
Interface1 = interface (IDispatch)
  [uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
  function Calculate(optional seed:Integer=0): Integer;
  procedure Reset;
  procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
  function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid ‘{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}’,version 1.0]
interface Interface1 :IDispatch
{
  long Calculate([in, optional, defaultvalue(0)] long seed);
  void Reset(void);
  [propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
  [propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

Dispatch interface syntax

The Object Pascal syntax for declaring dispinterface type information has the form

```
dispinterfacename = dispinterface [attributes]
functionlist
[propertylist]
end;
```

For example, the following text declares a dispinterface with the same methods and property as the previous interface:

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring 'dispatch interface for MyObj']
  function Calculate(seed:Integer): Integer [dispid 1];
  procedure Reset [dispid 2];
  property Range: Integer [dispid 3];
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring "dispatch interface for MyObj"]
dispinterface Interface1
{
  methods:
    [id(1)] int Calculate([in] int seed);
    [id(2)] void Reset(void);
  properties:
    [id(3)] int Value;
};
```

CoClass syntax

The Object Pascal syntax for declaring CoClass type information has the form

```
classname = coclass(interfacename[interfaceattributes], ...); [attributes];
```

For example, the following text declares a coclass for the interface *IMyInt* and dispinterface *DmyInt*:

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring 'A class',
  appobject]
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring 'A class',
  appobject]
coclass myapp
{
  methods:
  [source] interface IMyInt);
  dispinterface DMyInt;
};
```

Enum syntax

The Object Pascal syntax for declaring Enum type information has the form

```
enumname = ([attributes] enumlist);
```

For example, the following text declares an enumerated type with three values:

```
location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
         helpstring 'location of booth']
  Inside = 1 [helpstring 'Inside the pavillion'];
  Outside = 2 [helpstring 'Outside the pavillion'];
  Offsite = 3 [helpstring 'Not near the pavillion'];);
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring 'location of booth']
typedef enum
{
  [helpstring 'Inside the pavillion'] Inside = 1,
  [helpstring 'Outside the pavillion'] Outside = 2,
  [helpstring 'Not near the pavillion'] Offsite = 3
} location;
```

Alias syntax

The Object Pascal syntax for declaring Alias type information has the form

```
aliasname = basetype[attributes];
```

For example, the following text declares DWORD as an alias for integer:

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Record syntax

The Object Pascal syntax for declaring Record type information has the form

```
recordname = record [attributes] fieldlist end;
```

For example, the following text declares a record:

```
Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
               helpstring 'Task description']
  ID: Integer;
  StartDate: TDate;
  EndDate: TDate;
  Ownername: WideString;
  Subtasks: safearray of Integer;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring 'Task description']
typedef struct
{
  long ID;
  DATE StartDate;
  DATE EndDate;
  BSTR Ownername;
  SAFEARRAY (int) Subtasks;
} Tasks;
```

Union syntax

The Object Pascal syntax for declaring Union type information has the form

```
unionname = record [attributes]
case Integer of
  0: field1;
  1: field2;
  ...
end;
```

For example, the following text declares a union:

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                  helpstring 'item description']
case Integer of
  0: (Name: WideString);
  1: (ID: Integer);
  3: (Value: Double);
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring 'item description']
typedef union
{
  BSTR Name;
  long ID;
  double Value;
  } MyUnion;
```

Module syntax

The Object Pascal syntax for declaring Module type information has the form

```
modulename = module constants entrypoints end;
```

For example, the following text declares the type information for a module:

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                   dllname 'circle.dll']
  PI: Double = 3.14159;
  function area(radius: Double): Double [ entry 1 ]; stdcall;
  function circumference(radius: Double): Double [ entry 2 ]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
       dllname("circle.dll")]
module MyModule
{
  double PI = 3.14159;
  [entry(1)] double _stdcall area([in] double radius);
  [entry(2)] double _stdcall circumference([in] double radius);
};
```

Creating a new type library

You may want to create a type library that is independent of a particular COM object. For example, you might want to define a type library that contains type definitions that you use in several other type libraries. You can then create a type library of basic definitions and add it to the uses page of other type libraries.

You can also create a type library for an object that is not yet implemented. Once the type library contains the interface definition, you can use the COM object wizard to generate a CoClass and implementation.

To create a new type library,

1 Choose File | New | Other to open the New Items dialog box.

2 Choose the ActiveX page.

3 Select the Type Library icon.

4 Choose OK.

   The Type Library editor opens with a prompt to enter a name for the type library.

5 Enter a name for the type library. Continue by adding elements to your type library.

Opening an existing type library

When you use the wizards to create an ActiveX control, Automation object, Active form, Active Server Page object, COM object, transactional object, remote data module, or transactional data module, a type library is automatically created with an implementation unit. In addition, you may have type libraries that are associated with other products (servers) that are available on your system.

To open a type library that is not currently part of your project,

1 Choose File | Open from the main menu in the IDE.

2 In the Open dialog box, set the File Type to type library.

3 Navigate to the desired type library files and choose Open.

To open a type library associated with the current project,

1 Choose View | Type Library.

Note When you use the CORBA Object wizard, you can also choose View | Type Library to edit the CORBA Object interfaces. What you see is not, technically speaking, a type library, but you can use it in much the same way.

Now, you can add interfaces, CoClasses, and other elements of the type library such as enumerations, properties, and methods.

Note Changes you make to any type library information with the Type Library editor can be automatically reflected in the associated implementation class. If you want to review the changes before they are added, be sure that the Apply Updates dialog is on. It is on by default and can be changed in the setting, "Display updates before refreshing," on the Tools | Environment Options | Type Library page. For more information, see "Apply Updates dialog" on page 34-25.

Tip When writing client applications, you do not need to open the type library. You only need the *Project*_TLB unit that the Type Library editor creates from a type library, not the type library itself. You can add this file directly to a client project, or, if the type library is registered on your system, you can use the Import Type Library dialog (Project | Import Type Library).

Adding an interface to the type library

To add an interface,

1. On the toolbar, click on the interface icon.

   An interface is added to the object list pane prompting you to add a name.

2. Type a name for the interface.

The new interface contains default attributes that you can modify as needed.

You can add properties (represented by getter/setter functions) and methods to suit the purpose of the interface.

Modifying an interface using the type library

There are several ways to modify an interface or dispinterface once it is created.

- You can change the interface's attributes using the page of type information that contains the information you want to change. Select the interface in the object list pane and then use the controls on the appropriate page of type information. For example, you may want to change the parent interface using the attributes page, or use the flags page to change whether or not it is a dual interface.
- You can edit the interface declaration directly by selecting the interface in the object list pane and then editing the declarations on the Text page.
- You can Add properties and methods to the interface (see the next section).
- You can modify the properties and methods already in your interface by changing their type information.
- You can associate it with a CoClass by selecting the CoClass in the object list pane, right-clicking on the Implements page, and choosing Insert Interface.

Note When using the type library to add a CORBA interface, most of the information on the attributes page is irrelevent. You will also not need the Flags page.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the Type Library editor to apply your changes to the implementation file by clicking the Refresh button on the toolbar. If you have the Apply Updates dialog enabled, the Type Library editor notifies you before updating the sources and warns you of potential problems. For example, if you rename an event interface by mistake, you may get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You also get a TODO comment in your source file immediately above it.

Warning If you ignore this warning and TODO comment, the code will not compile.

Adding properties and methods to an interface or dispinterface

To add properties or methods to an interface or dispinterface,

1. Select the interface, and choose either a property or method icon from the toolbar. If you are adding a property, you can click directly on the property icon to create a read/write property (with both a getter and a setter), or click the down arrow to display a menu of property types.

   The property access method members or method member is added to the object list pane, prompting you to add a name.

2. Type a name for the member.

The new member contains default settings on its attributes, parameters, and flags pages that you can modify to suit the member. For example, you will probably want to assign a type to a property on the attributes page. If you are adding a method, you will probably want to specify its parameters on the parameters page.

As an alternate approach, you can add properties and methods by typing directly into the text page using Pascal or IDL syntax. For example, if you are working in Pascal syntax, you can type the following property declarations into the text page of an interface:

```
Interface1 = interface(IDispatch)
  [ uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
    version 1.0,
    dual,
    oleautomation ]
 function AutoSelect: Integer [propget, dispid $00000002]; safecall; // Add this
 function AutoSize: WordBool [propget, dispid $00000001]; safecall; // And this
 procedure AutoSize(Value: WordBool) [propput, dispid $00000001]; safecall; // And this
end;
```

If you are working in IDL, you can add the same declarations as follows:

```
[
  uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
  version(1.0),
  dual,
  oleautomation
]
interface Interface1: IDispatch
{ // Add everything between the curly braces
[propget, id(0x00000002)]
  HRESULT _stdcall AutoSelect([out, retval] long Value );
  [propget, id(0x00000003)]
  HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
  [propput, id(0x00000003)]
  HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

After you have added members to an interface using the interface text page, the members appear as separate items in the object list pane, each with its own attributes, flags, and parameters pages. You can modify each new property or method by selecting it in the object list pane and using these pages, or by making edits directly in the text page.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the Type Library editor to apply your changes to the implementation file by clicking the Refresh button on the toolbar. The Type Library editor adds new methods to your implementation class to reflect the new members. You can then locate the new methods in implementation unit's source code and fill in their bodies to complete the implementation.

If you have the Apply Updates dialog enabled, the Type Library editor notifies you of all changes before updating the sources and warns you of potential problems.

Adding a CoClass to the type library

The easiest way to add a CoClass to your project is to choose File | New | Other from the main menu in the IDE and use the appropriate wizard on the ActiveX or Multitier page of the New Items dialog. The advantage to this approach is that, in addition to adding the CoClass and its interface to the type library, the wizard adds an implementation unit and updates the project file to include the new implementation unit in its uses clause.

If you are not using a wizard, however, you can create a CoClass by clicking the CoClass icon on the toolbar and then specifying its attributes. You will probably want to give the new CoClass a name (on the Attributes page), and may want to use the Flags page to indicate information such as whether the CoClass is an application object, whether it represents an ActiveX control, and so on.

Note When you add a CoClass to a type library using the toolbar instead of a wizard, you must generate the implementation for the CoClass yourself and update it by hand every time you change an element on one of the CoClass's interfaces. You can't add members directly to a CoClass. Instead, you implicitly add members when you add an interface to the CoClass.

Adding an interface to a CoClass

CoClasses are defined by the interfaces they present to clients. While you can add any number of properties and methods to the implementation class of a CoClass, clients can only see those properties and methods that are exposed by interfaces associated with the CoClass.

To associate an interface with a CoClass, right-click in the Implements page for the class and choose Insert Interface to display a list of interfaces from which you can choose. The list includes interfaces that are defined in the current type library and those defined in any type libraries that the current type library references. Choose an interface you want the class to implement. The interface is added to the page with its GUID and other attributes.

If the CoClass was generated by a wizard, the Type Library editor automatically updates the implementation class to include skeletal methods for the methods (including property access methods) of any interfaces you add this way.If you have the Apply Updates dialog enabled, the Type Library editor notifies you before updating the sources and warns you of potential problems.

Adding an enumeration to the type library

To add enumerations to a type library,

1 On the toolbar, click on the enum icon.

   An enum type is added to the object list pane prompting you to add a name.

2 Type a name for the enumeration.

   The new enum is empty and contains default attributes in its attributes page for you to modify.

Add values to the enum by clicking on the New Const button. Then, select each enumerated value and assign it a name (and possibly a value) using the attributes page.

Once you have added an enumeration, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the enumeration as the type for a property or parameter.

Adding an alias to the type library

To add an alias to a type library,

1 On the toolbar, click on the alias icon.

   An alias type is added to the object list pane prompting you to add a name.

2 Type a name for the alias.

   By default, the new alias stands for an Integer type. Use the Attributes page to change this to the type you want the alias to represent.

Once you have added an alias, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the alias as the type for a property or parameter.

Adding a record or union to the type library

To add a record or union to a type library,

1 On the toolbar, click on the record icon or the union icon.

The selected type element is added to the object list pane prompting you to add a name.

2 Type a name for the record or union.

At this point, the new record or union contains no fields.

3 With the record or union selected in the object list pane, click on the field icon in the toolbar. Specify the field's name and type, using the Attributes page.

4 Repeat step 3 for as many fields as you need.

Once you have defined the record or union, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the record or union as the type for a property or parameter.

Adding a module to the type library

To add a module to a type library,

1 On the toolbar, click on the module icon.

The selected module is added to the object list pane prompting you to add a name.

2 Type a name for the module.

3 On the Attributes page, specify the name of the DLL whose entry points the Module represents.

4 Add any methods from the DLL you specified in step 3 by clicking on the Method icon in the toolbar and then using the attributes pages to describe the method.

5 Add any constants you want the module to define by clicking on the Const icon on the toolbar. For each constant, specify a name, type, and value.

Saving and registering type library information

After modifying your type library, you'll want to save and register the type library information.

Saving the type library automatically updates:

- The binary type library file (.tlb extension).
- The *Project*_TLB unit that represents its contents
- The implementation code for any CoClasses that were generated by a wizard.

Note The type library is stored as a separate binary (.TLB) file, but is also linked into the server (.EXE, DLL, or .OCX).

Note When using the Type Library editor for CORBA interfaces, the *Project*_TLB.pas unit defines the stub and skeleton objects required by the CORBA application.

The Type Library editor gives you options for storing your type library information. Which way you choose depends on what stage you are at in implementing the type library:

- Save, to save both the .TLB and the *Project*_TLB unit to disk.
- Refresh, to update the type library units in memory only.
- Register, to add an entry for the type library in your system's Windows registry. This is done automatically when the server with which the .TLB is associated is itself registered.
- Export, to save a .IDL file that contains the type and interface definitions in IDL syntax.

All the above methods perform syntax checking. When you refresh, register, or save the type library, Delphi automatically updates the implementation unit of any CoClasses that were created using a wizard. Optionally, you can review these updates before they are committed, if you have the Type Library editor option, Apply Updates on.

Apply Updates dialog

The Apply Updates dialog appears when you refresh, register, or save the type library if you have selected "Display updates before refreshing' in the Tools | Environment Options | Type Library page (which is on by default).

Without this option, the Type Library editor automatically updates the sources of the associated object when you make changes in the editor. With this option, you have a chance to veto the proposed changes when you attempt to refresh, save, or register the type library.

The Apply Updates dialog will warn you about potential errors, and will insert TODO comments in your source file. For example, if you rename an event by mistake, you will get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You will also get a TODO comment in your source file immediately above it.

Note If you ignore this warning and TODO comment, the code will not compile.

Saving a type library

Saving a type library

- Performs a syntax and validity check.
- Saves information out to a .TLB file.
- Saves information out to the *Project*_TLB unit.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with a CoClass that was generated by a wizard.

To save the type library, choose File | Save from the Delphi main menu.

Refreshing the type library

Refreshing the type library

- Performs a syntax check.
- Regenerates the Delphi type library units in memory only. It does not save any files to disk.
- Notifies the IDE's module manager to update the implementation, if the type library is associated with a CoClass that was generated by a wizard.

To refresh the type library choose the Refresh icon on the Type Library editor toolbar.

Note If you have renamed items in the type library, refreshing the implementation may create duplicate entries. In this case, you must move your code to the correct entry and delete any duplicates. Similarly, if you delete items in the type library, refreshing the implementation does not remove them from CoClasses (under the assumption that you are merely removing them from visibility to clients). You must delete these items manually in the implementation unit if they are no longer needed.

Registering the type library

Typically, you do not need to explicitly register a type library because it is registered automatically when you register your COM server application (see "Registering a COM object" on page 36-16). However, when you create a type library using the Type Library wizard, it is not associated with a server object. In this case, you can register the type library directly using the toolbar.

Registering the type library,

- Performs a syntax check
- Adds an entry to the Windows Registry for the type library

To register the type library, choose the Register icon on the Type Library editor toolbar.

Exporting an IDL file

Exporting the type library,

- Performs a syntax check.
- Creates an IDL file that contains the type information declarations. This file describes the type information in either CORBA IDL or Microsoft IDL.

To export the type library, choose the Export icon on the Type Library editor toolbar.

Deploying type libraries

By default, when you have a type library that was created as part of an ActiveX or Automation server project, the type library is automatically linked into the .DLL, .OCX, or EXE as a resource.

You can, however, deploy your application with the type library as a separate .TLB, as Delphi maintains the type library, if you prefer.

Historically, type libraries for Automation applications were stored as a separate file with the .TLB extension. Now, typical Automation applications compile the type libraries into the .OCX or .EXE file directly. The operating system expects the type library to be the first resource in the executable (.DLL, .OCX, or .EXE) file.

When you make type libraries other than the primary project type library available to application developers, the type libraries can be in any of the following forms:

- A resource. This resource should have the type TYPELIB and an integer ID. If you choose to build type libraries with a resource compiler, it must be declared in the resource (.RC) file as follows:

  ```
  1 typelib mylib1.tlb
  2 typelib mylib2.tlb
  ```

  There can be multiple type library resources in an ActiveX library. Application developers use the resource compiler to add the .TLB file to their own ActiveX library.

- Stand-alone binary files. The .TLB file output by the Type Library editor is a binary file.

Chapter

35

Creating COM clients

COM clients are applications that make use of a COM object implemented by another application or library. The most common types are applications that control an Automation server (Automation controllers) and applications that host an ActiveX control (ActiveX containers).

At first glance these two types of COM client are very different: The typical Automation controller launches an external server EXE and issues commands to make that server perform tasks on its behalf. The Automation server is usually nonvisual and out-of-process. The typical ActiveX client, on the other hand, hosts a visual control, using it much the same way you use any control on the Component palette. ActiveX servers are always in-process servers.

However, the task of writing these two types of COM client is remarkably similar: The client application obtains an interface for the server object and uses its properties and methods. Delphi makes this particularly easy by letting you wrap the server CoClass in a component on the client, which you can even install on the Component palette. Samples of such component wrappers appear on two pages of the Component palette: sample ActiveX wrappers appear on the ActiveX page and sample Automation objects appear on the Servers page.

When writing a COM client, you must understand the interface that the server exposes to clients, just as you must understand the properties and methods of a component from the Component palette to use it in your application. This interface (or set of interfaces) is determined by the server application, and typically published in a type library. For specific information on a particular server application's published interfaces, you should consult that application's documentation.

Even if you do not choose to wrap a server object in a component wrapper and install it on the Component palette, you must make its interface definition available to your application. To do this, you can import the server's type information.

Note You can also query the type information directly using COM APIs, but Delphi provides no special support for this.

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. These are discussed in “Creating Clients for servers that do not have a type library” on page 35-15.

Importing type library information

To make information about the COM server available to your client application, you must import the information about the server that is stored in the server’s type library. Your application can then use the resulting generated classes to control the server object.

There are two ways to import type library information:

- You can use the Import Type Library dialog to import all available information about the server types, objects, and interfaces. This is the most general method, because it lets you import information from any type library and can optionally generate component wrappers for all creatable CoClasses in the type library that are not flagged as Hidden, Restricted, or PreDeclID.
- You can use the Import ActiveX dialog if you are importing from the type library of an ActiveX control. This imports the same type information, but only creates component wrappers for CoClasses that represent ActiveX controls.
- You can use the command line utility tlibimp.exe which provides additional configuration options not available from within the IDE.
- A type library generated using a wizard is automatically imported using the same mechanism as the import type library menu item.

Regardless of which method you choose to import type library information, the resulting dialog creates a unit with the name *TypeLibName*_TLB, where *TypeLibName* is the name of the type library. This file contains declarations for the classes, types, and interfaces defined in the type library. By including it in your project, those definitions are available to your application so that you can create objects and call their interfaces. This file may be recreated by the IDE from time to time; as a result, making manual changes to the file is not recommended.

In addition to adding type definitions to the *TypeLibName*_TLB unit, the dialog can also create VCL class wrappers for any CoClasses defined in the type library. When you use the Import Type Library dialog, these wrappers are optional. When you use the Import ActiveX dialog, they are always generated for all CoClasses that represent controls.

The generated class wrappers represent the CoClasses to your application, and expose the properties and methods of its interfaces. If a CoClass supports the interfaces for generating events (*IConnectionPointContainer* and *IConnectionPoint*), the VCL class wrapper creates an event sink so that you can assign event handlers for the events as simply as you can for any other component. If you tell the dialog to install the generated VCL classes on the Component palette, you can use the Object Inspector to assign property values and event handlers.

Note The Import Type Library dialog does not create class wrappers for COM+ event objects. To write a client that responds to events generated by a COM+ event object, you must create the event sink programmatically. This process is described in "Handling COM+ events" on page 35-14.

For more details about the code generated when you import a type library, see "Code generated when you import type library information" on page 35-5.

Using the Import Type Library dialog

To import a type library,

1 Choose Project | Import Type Library.
2 Select the type library from the list.

   The dialog lists all the libraries registered on this system. If the type library is not in the list, choose the Add button, find and select the type library file, choose OK. This registers the type library, making it available. Then repeat step 2. Note that the type library could be a stand-alone type library file (.tlb, .olb), or a server that provides a type library (.dll, .ocx, .exe).

3 If you want to generate a VCL component that wraps a CoClass in the type library, check Generate Component Wrapper. If you do not generate the component, you can still use the CoClass by using the definitions in the *TypeLibName*_TLB unit. However, you will have to write your own calls to create the server object and, if necessary, to set up an event sink.

   The Import Type Library dialog only imports CoClasses that are have the CanCreate flag set and that do not have the Hidden, Restricted, or PreDeclID flags set. These flags can be overridden using the command-line utility tlibimp.exe.

4 If you do not want to install a generated component wrapper on the Component palette, choose Create Unit. This generates the *TypeLibName*_TLB unit and, if you checked Generate Component Wrapper in step 3, adds the declaration of the component wrapper. This exits the Import Type Library dialog.
5 If you want to install the generated component wrapper on the Component palette, select the Palette page on which this component will reside and then choose Install. This generates the *TypeLibName*_TLB unit, like the Create Unit button, and then displays the Install component dialog, letting you specify the package where the components should reside (either an existing package or a new one). This button is grayed out if no component can be created for the type library.

When you exit the Import Type Library dialog, the new *TypeLibName*_TLB unit appears in the directory specified by the Unit dir name control. This file contains declarations for the elements defined in the type library, as well as the generated component wrapper if you checked Generate Component Wrapper.

In addition, if you installed the generated component wrapper, a server object that the type library described now resides on the Component palette. You can use the Object Inspector to set properties or write an event handler for the server. If you add the component to a form or data module, you can right-click on it at design time to see its property page (if it supports one).

Note The Servers page of the Component palette contains a number of example Automation servers that were imported this way for you.

Using the Import ActiveX dialog

To import an ActiveX control,

1 Choose Component | Import ActiveX Control.

2 Select the type library from the list.

The dialog lists all the registered libraries that define ActiveX controls. (This is a subset of the libraries listed in the Import Type Library dialog.) If the type library is not in the list, choose the Add button, find and select the type library file, choose OK. This registers the type library, making it available. Then repeat step 2. Note that the type library could be a stand-alone type library file (.tlb, .olb), or an ActiveX server (.dll, .ocx).

3 If you do not want to install the ActiveX control on the Component palette, choose Create Unit. This generates the *TypeLibName*_TLB unit and adds the declaration of its component wrapper. This exits the Import ActiveX dialog.

4 If you want to install the ActiveX control on the Component palette, select the Palette page on which this component will reside and then choose Install. This generates the *TypeLibName*_TLB unit, like the Create Unit button, and then displays the Install component dialog, letting you specify the package where the components should reside (either an existing package or a new one).

When you exit the Import ActiveX dialog, the new *TypeLibName*_TLB unit appears in the directory specified by the Unit dir name control. This file contains declarations for the elements defined in the type library, as well as the generated component wrapper for the ActiveX control.

Note Unlike the Import Type Library dialog where it is optional, the import ActiveX dialog always generates a component wrapper. This is because, as a visual control, an ActiveX control needs the additional support of the component wrapper so that it can fit in with VCL forms.

If you installed the generated component wrapper, an ActiveX control now resides on the Component palette. You can use the Object Inspector to set properties or write event handlers for this control. If you add the control to a form or data module, you can right-click on it at design time to see its property page (if it supports one).

Note The ActiveX page of the Component palette contains a number of example ActiveX controls that were imported this way for you.

Code generated when you import type library information

Once you import a type library, you can view the generated *TypeLibName*_TLB unit. At the top, you will find the following:

- Constant declarations giving symbolic names to the GUIDS of the type library and its interfaces and CoClasses. The names for these constants are generated as follows:
    - The GUID for the type library has the form LBID_*TypeLibName*, where *TypeLibName* is the name of the type library.
    - The GUID for an interface has the form IID_*InterfaceName*, where *InterfaceName* is the name of the interface.
    - The GUID for a dispinterface has the form DIID_*InterfaceName*, where *InterfaceName* is the name of the dispinterface.
    - The GUID for a CoClass has the form CLASS_*ClassName*, where *ClassName* is the name of the CoClass.
- Declarations for the CoClasses in the type library. These map each CoClass to its default interface.
- Declarations for the interfaces and dispinterfaces in the type library.
- Declarations for a creator class for each CoClass whose default interface supports Vtable binding. The creator class has two class methods, *Create* and *CreateRemote*, that can be used to instantiate the CoClass locally (*Create*) or remotely (*CreateRemote*).These methods return the default interface for the CoClass.

These declarations provide you with what you need to create instances of the CoClass and access its interface. All you need do is add the generated *TypeLibName*_TLB.pas file to the uses clause of the unit where you wish to bind to a CoClass and call its interfaces.

Note This portion of the *TypeLibName*_TLB unit is also generated when you use the Type Library editor or the command-line utility TLIBIMP.

If you want to use an ActiveX control, you also need the generated VCL wrapper in addition to the declarations described above. The VCL wrapper handles window management issues for the control. You may also have generated a VCL wrapper for other CoClasses in the Import Type Library dialog. These VCL wrappers simplify the task of creating server objects and calling their methods. They are especially recommended if you want your client application to respond to events.

The declarations for generated VCL wrappers appear at the bottom of the interface section. Component wrappers for ActiveX controls are descendants of *TOleControl*. Component wrappers for Automation objects descend from *TOleServer*. The generated component wrapper adds the properties, events, and methods exposed by the CoClass's interface. You can use this component like any other VCL component.

Warning You should not edit the generated *TypeLibName*_TLB unit. It is regenerated each time the type library is refreshed, so any changes will be overwritten.

Note For the most up-to-date information about the generated code, refer to the comments in the automatically-generated *TypeLibName*_TLB unit.

Controlling an imported object

After importing type library information, you are ready to start programming with the imported objects. How you proceed depends in part on the objects, and in part on whether you have chosen to create component wrappers.

Using component wrappers

If you generated a component wrapper for your server object, writing your COM client application is not very different from writing any other application that contains VCL components. The server object's properties, methods, and events are already encapsulated in the VCL component. You need only assign event handlers, set property values, and call methods.

To use the properties, methods, and events of the server object, see the documentation for your server. The component wrapper automatically provides a dual interface where possible. Delphi determines the VTable layout from information in the type library.

In addition, your new component inherits certain important properties and methods from its base class.

ActiveX wrappers

You should always use a component wrapper when hosting ActiveX controls, because the component wrapper integrates the control's window into the VCL framework.

The properties and methods an ActiveX control inherits from *TOleControl* allow you to access the underlying interface or obtain information about the control. Most applications, however, do not need to use these. Instead, you use the imported control the same way you would use any other VCL control.

Typically, ActiveX controls provide a property page that lets you set their properties. Property pages are similar to the component editors some components display when you double-click on them in the form designer. To display an ActiveX control's property page, right click and choose Properties.

The way you use most imported ActiveX controls is determined by the server application. However, ActiveX controls use a standard set of notifications when they represent the data from a database field. See "Using data-aware ActiveX controls" on page 35-8 for information on how to host such ActiveX controls.

Automation object wrappers

The wrappers for Automation objects let you control how you want to form the connection to your server object:

- The *ConnectKind* property indicates whether the server is local or remote and whether you want to connect to a server that is already running or if a new instance should be launched. When connecting to a remote server, you must specify the machine name using the *RemoteMachineName* property.
- Once you have specified the *ConnectKind*, there are three ways you can connect your component to the server:
  - you can explicitly connect to the server by calling the component's *Connect* method.
  - You can tell the component to connect automatically when your application starts up by setting the *AutoConnect* property to *True*.
  - You do not need to explicitly connect to the server. The component automatically forms a connection when you use one of the server's properties or methods using the component.

Calling methods or accessing properties is the same as using any other component:

```
TServerComponent1.DoSomething;
```

Handling events is easy, because you can use the Object Inspector to write event handlers. Note, however, that the event handler on your component may have slightly different parameters than those defined for the event in the type library. Specifically, pointer types (var parameters and interface pointers) are changed to Variants. You must explicitly cast var parameters to the underlying type before assigning a value. Interface pointers can be cast to the appropriate interface type using the **as** operator.For example, the following code shows an event handler for the ExcelApplication event, OnNewWorkBook. The event handler has a parameter that provides the interface of another CoClass (ExcelWorkbook). However, the interface is not passed as an ExcelWorkbook interface pointer, but rather as an OleVariant.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
  { Note how the OleVariant for the interface must be cast to the correct type }
  ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkbook));
end;
```

In this example, the event handler assigns the workbook to an ExcelWorkbook component (ExcelWorkbook1). This demonstrates how to connect a component wrapper to an existing interface by using the *ConnectTo* method. The *ConnectTo* method is added to the generated code for the component wrapper.

Servers that have an application object expose a Quit method on that object to let clients terminate the connection. Quit typically exposes functionality that is equivalent to using the File menu to quit the application. Code to call the Quit method is generated in your component's *Disconnect* method. If it is possible to call the Quit method with no parameters, the component wrapper also has an *AutoQuit* property. *AutoQuit* causes your controller to call Quit when the component is freed. If you want to disconnect at some other time, or if the Quit method requires parameters, you must call it explicitly. Quit appears as a public method on the generated component.

Using data-aware ActiveX controls

When you use a data-aware ActiveX control in a Delphi application, you must associate it with the database whose data it represents. To do this, you need a data source component, just as you need a data source for any data-aware VCL control.

After you place the data-aware ActiveX control in the form designer, assign its *DataSource* property to the data source that represents the desired dataset. Once you have specified a data source, you can use the Data Bindings editor to link the control's data-bound property to a field in the dataset.

To display the Data Bindings editor, right-click the data-aware ActiveX control to display a list of options. In addition to the basic   options, the additional Data Bindings item appears. Select this item to see the Data Bindings editor, which lists the names of fields in the dataset and the bindable properties of the ActiveX control.

To bind a field to a property,

1. In the ActiveX Data Bindings Editor dialog, select a field and a property name.

   Field Name lists the fields of the database and Property Name lists the ActiveX control properties that can be bound to a database field. The dispID of the property is in parentheses, for example, Value(12).

2. Click Bind and OK.

Note If no properties appear in the dialog, the ActiveX control contains no data-aware properties. To enable simple data binding for a property of an ActiveX control, use the type library as described in "Enabling simple data binding with the type library" on page 38-10.

The following example walks you through the steps of using a data-aware ActiveX control in the Delphi container. This example uses the Microsoft Calendar Control, which is available if you have Microsoft Office 97 installed on your system.

1. From the Delphi main menu, choose Component | Import ActiveX Control.
2. Select a data-aware ActiveX control, such as the Microsoft Calendar control 8.0, change its class name to *TCalendarAXControl*, and click Install.
3. In the Install dialog, click OK to add the control to the default user package, which makes the control available on the Palette.
4. Choose Close All and File | New | Application to begin a new application.
5. From the ActiveX tab, drop a *TCalendarAXControl* object, which you just added to the Palette, onto the form.
6. From the Data Access tab, drop a *DataSource* and *Table* object onto the form.
7. Select the *DataSource* object and set its *DataSet* property to *Table1*.
8. Select the *Table* object and do the following:
   - Set the *DatabaseName* property to DBDEMOS
   - Set the *TableName* property to EMPLOYEE.DB
   - Set the *Active* property to *True*

9 Select the *TCalendarAXControl* object and set its *DataSource* property to *DataSource1*.

10 Select the *TCalendarAXControl* object, right-click, and choose Data Bindings to invoke the ActiveX Control Data Bindings Editor.

Field Name lists all the fields in the active database. Property Name lists those properties of the ActiveX Control that can be bound to a database field. The dispID of the property is in parentheses.

11 Select the *HireDate* field and the *Value* property name, choose Bind, and OK.

The field name and property are now bound.

12 From the Data Controls tab, drop a *DBGrid* object onto the form and set its *DataSource* property to *DataSource1*.

13 From the Data Controls tab, drop a *DBNavigator* object onto the form and set its *DataSource* property to *DataSource1*.

14 Run the application.

15 Test the application as follows:

With the *HireDate* field displayed in the *DBGrid* object, navigate through the database using the Navigator object. The dates in the ActiveX control change as you move through the database.

Example: Printing a document with Microsoft Word

The following steps show how to create an Automation controller that prints a document using Microsoft Word 8 from Office 97.

Before you begin: Create a new project that consists of a form, a button, and an open dialog box (*TOpenDialog*). These controls constitute the Automation controller.

Step 1: Prepare Delphi for this example

For your convenience, Delphi has provided many common servers, such as Word, Excel, and PowerPoint, on the Component palette. To demonstrate how to import a server, we use Word. Since it already exists on the Component palette, this first step asks you to remove the package containing Word so that you can see how to install it on the palette. Step 4 describes how to return the Component palette to its normal state.

To remove Word from the Component palette,

1 Choose Component | Install packages.

2 Click Borland Sample Automation Server components and choose Remove.

The Servers page of the Component palette no longer contains any of the servers supplied with Delphi. (If no other servers have been imported, the Servers page also disappears.)

Step 2: Import the Word type library

To import the Word type library,

1 Choose Project|Import Type Library.

2 In the Import Type Library dialog,

   1 Select Microsoft Office 8.0 Object Library.

   If Word (Version 8) is not in the list, choose the Add button, go to Program Files\Microsoft Office\Office, select the Word type library file, MSWord8.olb choose Add, and then select Word (Version 8) from the list.

   2 For Palette Page, choose Servers.

   3 Choose Install.

   The Install dialog appears. Select the Into New Packages tab and type WordExample to create a new package containing this type library.

3 Go to the Servers Palette Page, select WordApplication and place it on a form.

4 Write an event handler for the button object as described in the next step.

Step 3: Use a VTable or dispatch interface object to control Microsoft Word

You can use either a VTable or a dispatch object to control Microsoft Word.

Using a VTable interface object

By dropping an instance of the WordApplication object onto your form, you can easily access the control using a VTable interface object. You simply call on methods of the class you just created. For Word, this is the *TWordApplication* class.

1 Select the button, double-click its *OnClick* event handler and supply the following event handling code:

```
procedure TForm1.Button1Click(Sender: TObject);

var
   FileName: OleVariant;
begin
 if OpenDialog1.Execute then
  begin
     FileName := OpenDialog1.FileName;

WordApplication1.Documents.Open(FileName,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam);

WordApplication1.ActiveDocument.PrintOut(
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam, EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam,EmptyParam,
   EmptyParam,EmptyParam);
   end;

end;
```

2 Build and run the program. By clicking the button, Word prompts you for a file to print.

Using a dispatch interface object

As an alternate, you can use a dispatch interface for late binding. To use a dispatch interface object, you create and initialize the Application object using the _ApplicationDisp dispatch wrapper class as follows. Notice that dispinterface methods are "documented" by the source as returning Vtable interfaces, but, in fact, you must cast them to dispatch interfaces.

1 Select the button, double-click its *OnQuit* event handler and supply the following event handling code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
MyWord : _ApplicationDisp;
FileName : OleVariant;
begin
  if OpenDialog1.Execute then
  begin
    FileName := OpenDialog1.FileName;
    MyWord := CoWordApplication.Create as
        _ApplicationDisp;
    (MyWord.Documents as DocumentsDisp).Open(FileName,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam);
    (MyWord.ActiveDocument as _DocumentDisp).PrintOut(EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
        EmptyParam,EmptyParam,EmptyParam);
    MyWord.Quit(EmptyParam,EmptyParam,EmptyParam);
  end;
end;
```

2 Build and run the program. By clicking the button, Word prompts you for a file to print.

Step 4: Clean up the example

After completing this example, you will want to restore Delphi to its original form.

1 Delete the objects on this Servers page:

- Choose Component | Install Packages.
- From the list, select the WordExample package and click remove.
- Click Yes to the message box asking for confirmation.
- Exit the Install Packages dialog by clicking OK.

2 Return the Borland Sample Automation Server Components package:

- Choose Component | Install Packages.
- Click the Add button.

- In the resulting dialog, choose dclaxserver50.bpl.
- Exit the Install Packages dialog by clicking OK.

Writing client code based on type library definitions

Although you must use a component wrapper for hosting an ActiveX control, you can write an Automation controller using only the definitions from the type library that appear in the *TypeLibName*_TLB unit. This process is a bit more involved that letting a component do the work, especially if you need to respond to events.

Connecting to a server

Before you can drive an Automation server from your controller application, you must obtain a reference to an interface it supports. Typically, you connect to a server through its main interface. For example, you connect to Microsoft Word through the WordApplication component.

If the main interface is a dual interface, you can use the creator objects in the *TypeLibName*_TLB.pas file. The creator classes have the same name as the CoClass, with the prefix "Co" added. You can connect to a server on the same machine by calling the *Create* method, or a server on a remote machine using the *CreateRemote* method. Because *Create* and *CreateRemote* are class methods, you do not need an instance of the creator class to call them.

```
MyInterface := CoServerClassName.Create;
MyInterface := CoServerClassName.CreateRemote('Machine1');
```

Create and *CreateRemote* return the default interface for the CoClass.

If the default interface is a dispatch interface, then there is no Creator class generated for the CoClass. Instead, you can call the global *CreateOleObject* function, passing in the GUID for the CoClass (there is a constant for this GUID defined at the top of the _TLB unit). *CreateOleObject* returns an IDispatch pointer for the default interface.

Controlling an Automation server using a dual interface

After using the automatically generated creator class to connect to the server, you call methods of the interface. For example,

```
var
  MyInterface : _Application;
begin
  MyInterface := CoWordApplication.Create;
  MyInterface.DoSomething;
```

The interface and creator class are defined in the *TypeLibName*_TLB unit that is generated automatically when you import a type library.

For information about dual interfaces, see "Dual interfaces" on page 36-13.

Controlling an Automation server using a dispatch interface

Typically, you use the dual interface to control the Automation server, as described above. However, you may find a need to control an Automation server with a dispatch interface because no dual interface is available.

To call the methods of a dispatch interface,

1. Connect to the server, using the global *CreateOleObject* function.
2. Use the **as** operator to cast the *IDispatch* interface returned by *CreateOleObject* to the dispinterface for the CoClass. This dispinterface type is declared in the *TypeLibName*_TLB unit.
3. Control the Automation server by calling methods of the dispinterface.

   Another way to use dispatch interfaces is to assign them to a *Variant*. By assigning the interface returned by *CreateOleObject* to a *Variant*, you can take advantage of the *Variant* type's built-in support for interfaces. Simply call the methods of the interface, and the *Variant* automatically handles all *IDispatch* calls, fetching the dispatch ID and invoking the appropriate method. The Variant type includes built-in support for calling dispatch interfaces, through its **var.**

```
  V: Variant;
begin
  V:= CreateOleObject('TheServerObject');
  V.MethodName; { calls the specified method }
  ...
```

An advantage of using *Variants* is that you do not need to import the type library, because *Variants* use only the standard *IDispatch* methods to call the server. The trade-off is that *Variants* are slower, because they use dynamic binding at runtime.

For more information on dispatch interfaces, see “Automation interfaces” on page 36-12.

Handling events in an automation controller

When you generate a Component wrapper for an object whose type library you import, you can respond to events simply using the events that are added to the generated component. If you do not use a Component wrapper, however, (or if the server uses COM+ events), you must write the event sink code yourself.

Handling Automation events programmatically

Before you can handle events, you must define an event sink. This is a class that implements the event dispatch interface that is defined in the server's type library.

To write the event sink, create an object that implements the event dispatch interface:

```
TServerEventsSink = class(TObject, _TheServerEvents)
...{ declare the methods of _TheServerEvents here }
end;
```

Once you have an instance of your event sink, you must inform the server object of its existence so that the server can call it. To do this, you call the global *InterfaceConnect* procedure, passing it

- The interface to the server that generates events.
- The GUID for the event interface that your event sink handles.
- An IUnknown interface for your event sink.
- A variable that receives a Longint that represents the connection between the server and your event sink.

```
{MyInterface is the server interface you got when you connected to the server }
InterfaceConnect(MyInterface, DIID_TheServerEvents,
                 MyEventSinkObject as IUnknown, cookievar);
```

After calling *InterfaceConnect*, your event sink is connected and receives calls from the server when events occur.

You must terminate the connection before you free your event sink. To do this, call the global *InterfaceDisconnect* procedure, passing it all the same parameters except for the interface to your event sink (and the final parameter is ingoing rather than outgoing):

```
InterfaceDisconnect(MyInterface, DIID_TheServerEvents, cookievar);
```

Note You must be certain that the server has released its connection to your event sink before you free it. Because you don't know how the server responds to the disconnect notification initiated by *InterfaceDisconnect*, this may lead to a race condition if you free your event sink immediately after the call. The easiest way to guard against problems is to have your event sink maintain its own reference count that is not decremented until the server releases the event sink's interface.

Handling COM+ events

Under COM+, servers use a special helper object to generate events rather than a set of special interfaces (*IConnectionPointContainer* and *IConnectionPoint*). Because of this, you can't use an event sink that descends from *TEventDispatcher. TEventDispatcher* is designed to work with those interfaces, not COM+ event objects.

Instead of defining an event sink, your client application defines a subscriber object. Subscriber objects, like event sinks, provide the implementation of the event interface. They differ from event sinks in that they subscribe to a particular event object rather than connecting to a server's connection point.

To define a subscriber object, use the COM Object wizard, selecting the event object's interface as the one you want to implement. The wizard generates an implementation unit with skeletal methods that you can fill in to create your event handlers. For more information about using the COM Object wizard to implement an existing interface, see "Using the COM object wizard" on page 36-2.

Note You may need to add the event object's interface to the registry using the wizard if it does not appear in the list of interfaces you can implement.

Once you create the subscriber object, you must subscribe to the event object's interface or to individual methods (events) on that interface. There are three types of subscriptions from which you can choose:

- **Transient subscriptions.** Like traditional event sinks, transient subscriptions are tied to the lifetime of an object instance. When the subscriber object is freed, the subscription ends and COM+ no longer forwards events to it.

- **Persistent subscriptions.** These are tied to the object class rather than a specific object instance. When the event occurs, COM locates or launches an instance of the subscriber object and calls its event handler. In-process objects (DLLs) use this type of subscription.
- **Per-user subscriptions.** These subscriptions provide a more secure version of transient subscriptions. Both the subscriber object and the server object that fires events must be running under the same user account on the same machine.

Note Objects that subscribe to COM+ events must be installed in a COM+ application.

Creating Clients for servers that do not have a type library

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. To write clients that host such objects, you can use the *TOleContainer* component. This component appears on the System page of the Component palette.

TOleContainer acts as a host site for an Ole2 object. It implements the *IOleClientSite* interface and, optionally, *IOleDocumentSite*. Communication is handled using OLE verbs.

To use *TOleContainer*,

1. Place a *TOleContainer* component on your form.
2. Set the *AllowActiveDoc* property to *True* if you want to host an Active document.
3. Set the *AllowInPlace* property to indicate whether the hosted object should appear in the *TOleContainer*, or in a separate window.
4. Write event handlers to respond when the object is activated, deactivated, moved, or resized.
5. To bind the *TOleContainer* object at design time, right click and choose Insert Object. In the Insert Object dialog, choose a server object to host.
6. To bind the *TOleContainer* object at runtime, you have several methods to choose from, depending on how you want to identify the server object. These include *CreateObject*, which takes a program id, *CreateObjectFromFile*, which takes the name of a file to which the object has been saved, *CreateObjectFromInfo*, which takes a record containing information on how to create the object, or *CreateLinkToFile*, which takes the name of a file to which the object was saved and links to it rather than embeds it.
7. Once the object is bound, you can access its interface using the *OleObjectInterface* property. However, because communication with Ole2 objects was based on OLE verbs, you will most likely want to send commands to the server using the *DoVerb* method.
8. When you want to release the server object, call the *DestroyObject* method.

Chapter

36

Creating simple COM servers

Delphi provides wizards to help you create various COM objects. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call.

Note COM servers and Automation is not available for use in CLX applications. This technology is for use on Windows only and is not cross-platform.

Two wizards, in particular, ease the process of creating simple COM objects:

- The COM Object wizard builds a lightweight COM object whose default interface descends from *IUnknown* or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create.
- The Automation Object wizard creates a simple Automation object whose default interface descends from *IDispatch*. *IDispatch* introduces a standard marshaling mechanism and support for late binding of interface calls.

Note COM defines many standard interfaces and mechanisms for handling specific situations. The Delphi wizards automate the most common tasks. However, some tasks, such as custom marshaling, are not supported by any Delphi wizards. For information on that and other technologies not explicitly supported by Delphi, refer to the Microsoft Developer's Network (MSDN) documentation. The Microsoft Web site also provides current information on COM support.

Overview of creating a COM object

Whether you use the Automation object wizard to create a new Automation server or the COM object wizard to create some other type of COM object, the process you follow is the same. It involves these steps:

1. Design the COM object.
2. Use the COM Object wizard or the Automation Object wizard to create the server object.
3. Define the interface that the object exposes to clients.
4. Register the COM object.
5. Test and debug the application.

Designing a COM object

When designing the COM object, you need to decide what COM interfaces you want to implement. You can write a COM object to implement an interface that has already been defined, or you can define a new interface for your object to implement. In addition, you can have your object support more than one interface. For information about standard COM interfaces that you might want to support, see the MSDN documentation.

- To create a COM object that implements an existing interface, use the COM Object wizard.
- To create a COM object that implements a new interface that you define, use either the COM Object wizard or the Automation Object wizard. The COM object wizard can generate a new default interface that descends from *IUnknown*, and the Automation object gives your object a default interface that descends from *IDispatch*. No matter which wizard you use, you can always use the Type Library editor later to change the parent interface of the default interface that the wizard generates.

In addition to deciding what interfaces to support, you must decide whether the COM object is an in-process server, out-of-process server, or remote server. For in-process servers and for out-of-process and remote servers that use a type library, COM marshals the data for you. Otherwise, you must consider how to marshal the data to out-of-process servers. For information on server types, see, “In-process, out-of-process, and remote servers,” on page 33-6.

Using the COM object wizard

The COM object wizard performs the following tasks:

- Creates a new unit.

- Defines a new class that descends from *TCOMObject* and sets up the class factory constructor. For more information on the base class, see “Code generated by wizards” on page 33-21.
- Optionally, adds a type library to your project and adds your object and its interface to the type library.

Before you create a COM object, create or open the project for the application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the COM object wizard,

1 Choose File | New | Other to open the New Items dialog box.

2 Select the tab labeled, ActiveX.

3 Double-click the COM object icon.

In the wizard, you must specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. The class created to implement your object has this name with a ‘T’ prepended. If you do not choose to implement an existing interface, the wizard gives your CoClass a default interface that has this name with an ‘I’ prepended.
- **Interface to implement:** By default, the wizard gives your object a default interface that descends from *IUnknown*. After exiting the wizard, you can then use the Type Library editor to add properties and methods to this interface. However, you can also select a pre-defined interface for your object to implement. Click the List button in the COM object wizard to bring up the Interface Selection wizard, where you can select any dual or custom interface defined in a type library registered on your system. The interface you select becomes the default interface for your new CoClass. The wizard adds all the methods on this interface to the generated implementation class, so that you only need to fill in the bodies of the methods in the implementation unit. Note that if you select an existing interface, the interface is not added to your project’s type library. This means that when deploying your object, you must also deploy the type library that defines the interface.
- **Instancing:** Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them. For information on the different possibilities, see “COM object instancing types” on page 36-5.
- **Threading Model**: Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on the different possibilities, see “Choosing a threading model” on page 36-6. For information on how to provide thread support to your application, see Chapter 9, “Writing multi-threaded applications.”

- **Type Library:** You can choose whether you want to include a type library for your object. This is recommended for two reasons: it lets you use the Type Library editor to define interfaces, thereby updating much of the implementation, and it gives clients an easy way to obtain information about your object and its interfaces. If you are implementing an existing interface, Delphi requires your project to use a type library. This is the only way to provide access to the original interface declaration. For information on type libraries, see “Type libraries” on page 33-15 and Chapter 34, “Working with type libraries”.
- **Marshaling:** If you have opted to create a type library and are willing to confine yourself to Automation-compatible types, you can let COM handle the marshaling for you when you are not generating an in-process server. By marking your object’s interface as OleAutomation in the type library, you enable COM to set up the proxies and stubs for you and handles passing parameters across process boundaries. For more information on this process, see “The marshaling mechanism” on page 33-8. You can only specify whether your interface is Automation-compatible if you are generating a new interface. If you select an existing interface, its attributes are already specified in its type library. If your object’s interface is not marked as OleAutomation, you must either create an in-process server or write your own marshaling code.

You can optionally add a description of your COM object. This description appears in the type library for your object if you create one.

Using the Automation object wizard

The Automation object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from *TAutoObject* and sets up the class factory constructor. For more information on the base class, see “Code generated by wizards” on page 33-21.
- Adds a type library to your project and adds your object and its interface to the type library.

Before you create an Automation object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.

To display the Automation wizard:

1. Choose File | New | Other.
2. Select the tab labeled, ActiveX.
3. Double-click the Automation Object icon.

In the wizard dialog, specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. Your object's default interface is created with a name based on this CoClass name with an 'I' prepended, and the class created to implement your object has this name with a 'T' prepended.
- **Instancing:** Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them. For information on the different possibilities, see "COM object instancing types" on page 36-5.
- **Threading Model**: Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on the different possibilities, see "Choosing a threading model" on page 36-6. For information on how to provide thread support to your application, see Chapter 9, "Writing multi-threaded applications."
- **Event support**: You must indicate whether you want your object to generate events to which clients can respond. The wizard can provide support for the interfaces required to generate events and the dispatching of calls to client event handlers. For information on how events work and what you need to do when implementing them, see "Exposing events to clients" on page 36-10.

You can optionally add a description of your COM object. This description appears in the type library for your object.

The Automation object implements a **dual interface**, which supports both early (compile-time) binding through the VTable and late (runtime) binding through the *IDispatch* interface. For more information, see "Dual interfaces" on page 36-13.

COM object instancing types

Many of the COM wizards require you to specify an instancing mode for the object. Instancing determines how many instances of your object clients can create in a single executable. If you specify a Single Instance model, for example, then COM once a client has instantiated your object, COM removes the application from view so that other clients must launch their own instances of the application. Because this affects the visibility of your application as a whole, the instancing mode must be consistent across all objects in your application that can be instantiated by clients. That is, you should not create one object in your application that uses Single Instance mode and another in the same application that uses Multiple Instance mode.

Note Instancing is ignored when your COM object is used only as an in-process server.

When the wizard creates a new COM object, it can have any of the following instancing types:

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly, although your application can create the object hand pass an interface for it to clients.
Single Instance	Allows clients to create only a single instance of the object for each executable (application), so creating multiple instances results in launching multiple instances of the application. Each client has its own dedicated instance of the server application. This option is commonly used for multiple document interface (MDI) applications.
Multiple Instances	Specifies that multiple clients can create instances of the object in the same process space. Any time a client requests service, a separate instance of the object is created. (That is, there can be multiple instances in a single executable.)

Choosing a threading model

When creating an object using a wizard, you select a threading model that your object agrees to support. By adding thread support to your COM object, you can improve its performance, because multiple clients can access your application at the same time.

Table 36.1 lists the different threading models you can specify.

Table 36.1 Threading models for COM objects

Threading model	Description	Implementation pros and cons
Single	The server provides no thread support. COM serializes client requests so that the application receives one request at a time.	Clients are handled one at a time so no threading support is needed. No performance benefit.
Apartment (or Single-threaded apartment)	COM ensures that only one client thread can call the object at a time. All client calls use the thread in which the object was created.	Objects can safely access their own instance data, but global data must be protected using critical sections or some other form of serialization. The thread's local variables are reliable across multiple calls. Some performance benefits.
Free (also called multi-threaded apartment)	Objects can receive calls on any number of threads at any time.	Objects must protect all instance and global data using critical sections or some other form of serialization. Thread local variables are *not* reliable across multiple calls.

Table 36.1 Threading models for COM objects (continued)

Threading model	Description	Implementation pros and cons
Both	This is the same as the Free-threaded model except that outgoing calls (for example, callbacks) are guaranteed to execute in the same thread.	Maximum performance and flexibility. Does not require the application to provide thread support for parameters supplied to outgoing calls.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict.	You must guard against thread conflicts involving global data and any instance data that is accessed by multiple methods. This model should not be used with objects that have a user interface (visual controls). This model is only available under COM+. Under COM, it is mapped to the Apartment model.

Note Local variables (except those in callbacks) are always safe, regardless of the threading model. This is because local variables are stored on the stack and each thread has its own stack. Local variables may not be safe in callbacks when using free-threading.

The threading model you choose in the wizard determines how the object is registered in the system Registry. You must make sure that your object implementation adheres to the threading model you have chosen. For general information on writing thread-safe code, see Chapter 9, “Writing multi-threaded applications.”

For in-process servers, setting the threading model in the wizard sets the threading model key in the CLSID registry entry.

Out-of-process servers are registered as EXE, and Delphi initializes COM for the highest threading model required. For example, if an EXE includes a free-threaded object, it is initialized for free threading, which means that it can provide the expected support for any free-threaded or apartment-threaded objects contained in the EXE. To manually override threading behavior in EXEs, use the CoInitFlags variable, which is described in the online help.

Writing an object that supports the free threading model

Use the free threading (or both) model rather than apartment threading whenever the object needs to be accessed from more than one thread. A common example is a client application connected to an object on a remote machine. When the remote client calls a method on that object, the server receives the call on a thread from the thread pool on the server machine. This receiving thread makes the call locally to the actual object; and, because the object supports the free threading model, the thread can make a direct call into the object.

If the object supported the apartment threading model instead, the call would have to be transferred to the thread on which the object was created, and the result would have to be transferred back into the receiving thread before returning to the client. This approach requires extra marshaling.

To support free threading, you must consider how instance data can be accessed for *each* method. If the method is writing to instance data, you must use critical sections or some other form of serialization, to protect the instance data. Likely, the overhead of serializing critical calls is less than executing COM's marshaling code.

Note that if the instance data is read-only, serialization is not needed.

Free-threaded in-process servers can improve performance by acting as the outer object in an aggregation with the free-threaded marshaler. The free-threaded marshaler provides a shortcut for COM's standard thread handling when a free-threaded DLL is called by a host (client) that is not free-threaded.

To aggregate with the free threaded marshaler, you must

- Call *CoCreateFreeThreadedMarshaler*, passing your object's *IUnknown* interface for the resulting free-threaded marshaler to use:

  ```
  CoCreateFreeThreadedMarshaler(self as IUnknown, FMarshaler);
  ```

  This line assigns the interface for the free-threaded marshaler to a class member, *FMarshaler*.
- Using the Type Library editor, add the *IMarshal* interface to the set of interfaces your CoClass implements.
- In your object's *QueryInterface* method, delegate calls for IDD_IMarshal to the free-threaded marshaler (stored as *FMarshaler* above).

Warning The free-threaded marshaler violates the normal rules of COM marshaling to provide additional efficiency. It should be used with care. In particular, it should only be aggregated with free-threaded objects in in-process servers, and should only be instantiated by the object that uses it (not another thread).

Writing an object that supports the apartment threading model

To implement the (single-threaded) apartment threading model, you must follow a few rules:

- The first thread in the application that gets created is COM's main thread. This is typically the thread on which WinMain was called. This must also be the last thread to uninitialize COM.
- Each thread in the apartment threading model must have a message loop, and the message queue must be checked frequently.
- When a thread gets a pointer to a COM interface, that pointer may only be used in that thread.

The single-threaded apartment model is the middle ground between providing no threading support and full, multi-threading support of the free threading model. A server committing to the apartment model promises that the server has serialized access to all of its global data (such as its object count). This is because different objects may try to access the global data from different threads. However, the object's instance data is safe because the methods are always called on the same thread.

Typically, controls for use in Web browsers use the apartment threading model because browser applications always initialize their threads as apartment.

Writing an object that supports the neutral threading model

Under COM+, you can use another threading model that is in between free threading and apartment threading: the neutral model. Like the free-threading model, this model allows multiple threads to access your object at the same time. There is no extra marshaling to transfer to the thread on which the object was created. However, your object is guaranteed to receive no conflicting calls.

Writing an object that uses the neutral threading model follows much the same rules as writing an apartment-threaded object, except that you do need to guard instance data against thread conflicts if it can be accessed by different methods in the object's interface. Any instance data that is only accessed by a single interface method is automatically thread-safe.

Defining a COM object's interface

When you use a wizard to create a COM object, the wizard automatically generates a type library (unless you specify otherwise in the COM object wizard). The type library provides a way for host applications to find out what the object can do. It also lets you define your object's interface using the Type Library editor. The interfaces you define in the Type Library editor define what properties, methods, and events your object exposes to clients.

Note If you selected an existing interface in the COM object wizard, you do not need to add properties and methods. The definition of the interface is imported from the type library in which it was defined. Instead, simply locate the methods of the imported interface in the implementation unit and fill in their bodies.

Adding a property to the object's interface

When you add a property to your object's interface using the Type Library editor, it automatically adds a method to read the property's value and/or a method to set the property's value. The Type Library editor, in turn, adds these methods to your implementation class, and in your implementation unit creates empty method implementations for you to complete.

To add a property to your object's interface,

1 In the type library editor, select the default interface for the object.

The default interface should be the name of the object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."

2. To expose a read/write property, click the Property button on the toolbar; otherwise, click the arrow next to the Property button on the toolbar, and then click the type of property to expose.
3. In the Attributes pane, specify the name and type of the property.
4. On the toolbar, click the Refresh button.

   A definition and skeletal implementations for the property access methods are inserted into the object's implementation unit.
5. In the implementation unit, locate the access methods for the property. These have names of the form Get_PropertyName and Set_PropertyName. Add code that gets or sets the property value of your object. This code may simply call an existing function inside the application, access a data member that you add to the object definition, or otherwise implement the property.

Adding a method to the object's interface

When you add a method to your object's interface using the Type Library editor, the Type Library editor can, in turn, add the methods to your implementation class, and in your implementation unit create empty implementation for you to complete.

To expose a method via your object's interface,

1. In the Type Library editor, select the default interface for the object.

   The default interface should be the name of the object preceded by the letter "I". To determine the default, in the Type Library editor, choose the CoClass and Implements tab, and check the list of implemented interfaces for the one marked, "Default."
2. Click the Method button.
3. In the Attributes pane, specify the name of the method.
4. In the Parameters pane, specify the method's return type and add the appropriate parameters.
5. On the toolbar, click the Refresh button.

   A definition and skeletal implementation for the method is inserted into the object's implementation unit.
6. In the implementation unit, locate the newly inserted method implementation. The method is completely empty. Fill in the body to perform whatever task the method represents.

Exposing events to clients

There are two types of events that a COM object can generate: traditional events and COM+ events.

- COM+ events require that you create a separate event object using the event object wizard and add code to call that event object from your server object. For more information about generating COM+ events, see "Generating events under COM+" on page 39-18.

- You can use the wizard to handle much of the work in generating traditional events. This process is described below.

Note The COM object wizard does not generate event support code. If you want your object to generate traditional events, you should use the Automation object wizard.

In order for an object to generate events, you need to do the following:

1. In the Automation wizard, check the box, Generate event support code.

   The wizard creates an object that includes an Events interface as well as the default interface. This Events interface has a name of the form I*CoClassname*Events. It is an outgoing (source) interface, which means that it is not an interface your object implements, but rather is an interface that clients must implement and which your object calls. (You can see this by selecting your CoClass, going to the Implements page, and noting that the Source column on the Events interface says *True*.)

   In addition to the Events interface, the wizard adds the *IConnectionPointContainer* interface to the declaration of your implementation class, and adds several class members for handling events. Of these new class members, the most important are *FConnectionPoint* and *FConnectionPoints*, which implement the *IConnectionPoint* and *IConnectionPointContainer* interfaces using built-in VCL classes. *FConnectionPoint* is maintained by another method that the wizard adds, *EventSinkChanged*.

2. In the Type Library editor, select the outgoing Events interface for your object. (This is the one with a name of the form I*CoClassName*Events)
3. Click the Method button from the Type Library toolbar. Each method you add to the Events interface represents an event handler that the client must implement.
4. In the Attributes pane, specify the name of the event handler, such as MyEvent.
5. On the toolbar, click the Refresh button.

   Your object implementation now has everything it needs to accept client event sinks and maintain a list of interfaces to call when the event occurs. To call these interfaces, you can create a method to generate each event on clients.

6. In the Code Editor, add a method to your object for firing each event. For example,

```
unit ev;
interface
uses
   ComObj, AxCtrls, ActiveX, Project1_TLB;
type
   TMyAutoObject = class (TAutoObject,IConnectionPointContainer, IMyAutoObject)
private
   .
   .
   .
public
   procedure Initialize; override;
   procedure Fire_MyEvent; { Add a method to fire the event}
```

7 Implement the method you added in the last step so that it iterates through all the event sinks maintained by your object's *FConnectionPoint* member:

```
procedure TMyAutoObject.Fire_MyEvent;
var
  I: Integer;
  EventSinkList: TList;
  EventSink: IMyAutoObjectEvents;
begin
  if FConnectionPoint <> nil then
  begin
    EventSinkList :=FConnectionPoint.SinkList; {get the list of client sinks }
    for I := 0 to EventSinkList.Count - 1 do
    begin
      EventSink := IUnknown(FEvents[I]) as IMyAutoObjectEvents;
      EventSink.MyEvent;
    end;
  end;
end;
```

8 Whenever you need to fire the event so that clients are informed of its occurrence, call the method that dispatches the event to all event sinks:

```
if EventOccurs then Fire_MyEvent; { Call method you created to fire events.}
```

Managing events in your Automation object

For a server to support traditional COM events, it must provide the definition of an outgoing interface which is implemented by a client. This outgoing interface includes all the event handlers the client must implement to respond to server events.

When a client has implemented the outgoing event interface, it registers its interest in receiving event notification by querying the server's *IConnectionPointContainer* interface. The *IConnectionPointContainer* interface returns the server's *IConnectionPoint* interface, which the client then uses to pass the server a pointer to its implementation of the event handlers (known as a sink).

The server maintains a list of all client sinks and calls methods on them when an event occurs, as described above.

Automation interfaces

The Automation Object wizard implements a dual interface by default, which means that the Automation object supports both

- Late binding at runtime, which is through the *IDispatch* interface. This is implemented as a dispatch interface, or **dispinterface**.
- Early binding at compile-time, which is accomplished through directly calling one of the member functions in the object's virtual function table (VTable). This is referred to as a **custom interface**.

Note Any interfaces generated by the COM object wizard that do not descend from *IDispatch* only support VTable calls.

Dual interfaces

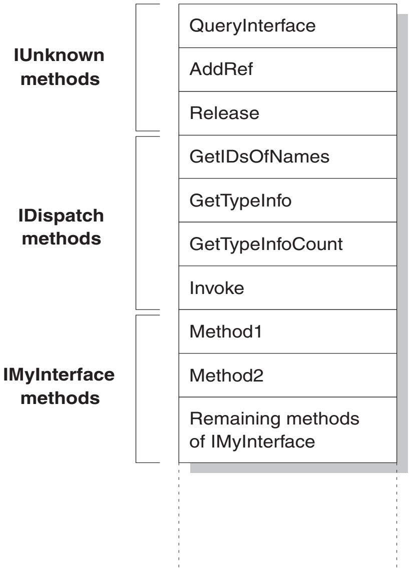
A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch*. For those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time binding, the more efficient VTable interface is used.

Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces:

- For VTable interfaces, the compiler performs type checking and provides more informative error messages.
- For Automation controllers that cannot obtain type information, the dispinterface provides runtime access to the object.
- For in-process servers, you have the benefit of fast access through VTable interfaces.
- For out-of-process servers, COM marshals data for both VTable interfaces and dispinterfaces. COM provides a generic proxy/stub implementation that can marshal the interface based on the information contained in a type library. For more information on marshaling, see, "Marshaling data," on page 36-15.

The following diagram depicts the *IMyInterface* interface in an object that supports a dual interface named *IMyInterface.* The first three entries of the VTable for a dual interface refer to the *IUnknown* interface, the next four entries refer to the *IDispatch* interface, and the remaining entries are COM entries for direct access to members of the custom interface.

Figure 36.1 Dual interface VTable



Dispatch interfaces

Automation controllers are clients that use the COM *IDispatch* interface to access the COM server objects. The controller must first create the object, then query the object's *IUnknown* interface for a pointer to its *IDispatch* interface. *IDispatch* keeps track of methods and properties internally by a dispatch identifier (dispID), which is a unique identification number for an interface member. Through *IDispatch*, a controller retrieves the object's type information for the dispatch interface and then maps interface member names to specific dispIDs. These dispIDs are available at runtime, and controllers get them by calling the *IDispatch* method, *GetIDsOfNames*.

Once it has the dispID, the controller can then call the *IDispatch* method, *Invoke*, to execute the appropriate code (property or method), packaging the parameters for the property or method into one of the *Invoke* parameters. *Invoke* has a fixed compile-time signature that allows it to accept any number of arguments when calling an interface method.

The Automation object's implementation of *Invoke* must then unpackage the parameters, call the property or method, and be prepared to handle any errors that occur. When the property or method returns, the object passes its return value back to the controller.

This is called late binding because the controller binds to the property or method at runtime rather than at compile time.

Note When importing a type library, Delphi will query for dispIDs at the time it generates the code, thereby allowing generated wrapper classes to call *Invoke* without calling *GetIDsOfNames*. This can significantly increase the runtime performance of controllers.

Custom interfaces

Custom interfaces are user-defined interfaces that allow clients to invoke interface methods based on their order in the VTable and knowledge of the argument types. The VTable lists the addresses of all the properties and methods that are members of the object, including the member functions of the interfaces that it supports. If the object does not support *IDispatch*, the entries for the members of the object's custom interfaces immediately follow the members of *IUnknown*.

If the object has a type library, you can access the custom interface through its VTable layout, which you can get using the Type Library editor. If the object has a type library and also supports *IDispatch*, a client can also get the dispIDs of the *IDispatch* interface and bind directly to a VTable offset. Delphi's type library importer (TLIBIMP) retrieves dispIDs at import time, so clients that use dispinterfaces can avoid calls to *GetIDsOfNames*; this information is already in the _TLB unit. However, clients still need to call *Invoke*.

Marshaling data

For out-of-process and remote servers, you must consider how COM marshals data outside the current process. You can provide marshaling:

- Automatically, using the *IDispatch* interface.
- Automatically, by creating a type library with your server and marking the interface with the OLE Automation flag. COM knows how to marshal all the **Automation-compatible** types in the type library and can set up the proxies and stubs for you. Some type restrictions apply to enable automatic marshaling.
- Manually by implementing all the methods of the *IMarshal* interface. This is called **custom marshaling**.

Note The first method (using *IDispatch*) is only available on Automation servers. The second method is automatically available on all objects that are created by wizards and which use a type library.

Automation compatible types

Function result and parameter types of the methods declared in dual and dispatch interfaces and interfaces that you mark as OLE Automation must be *Automation-compatible* types. The following types are OLE Automation-compatible:

- The predefined valid types such as *Smallint*, *Integer*, *Single*, *Double*, *WideString*. For a complete list, see “Valid types” on page 34-11.
- Enumeration types defined in a type library. OLE Automation-compatible enumeration types are stored as 32-bit values and are treated as values of type *Integer* for purposes of parameter passing.
- Interface types defined in a type library that are OLE Automation safe, that is, derived from *IDispatch* and containing only OLE Automation compatible types.
- Dispinterface types defined in a type library.
- Any custom record type defined within the type library.
- *IFont*, *IStrings*, and *IPicture*. Helper objects must be instantiated to map
  - an *IFont to* a *TFont*
  - an *IStrings* to a *TStrings*
  - an *IPicture* to a *TPicture*

  The ActiveX control and ActiveForm wizards create these helper objects automatically when needed. To use the helper objects, call the global routines, *GetOleFont*, *GetOleStrings*, *GetOlePicture*, respectively.

Type restrictions for automatic marshaling

For an interface to support automatic marshaling (also called Automation marshaling or type library marshaling), the following restrictions apply. When you edit your object using the type library editor, the editor enforces these restrictions:

- Types must be compatible for cross-platform communication. For example, you cannot use data structures (other than implementing another property object), unsigned arguments, AnsiStrings, and so on.
- String data types must be transferred as wide strings (BSTR). PChar and AnsiString cannot be marshaled safely.
- All members of a dual interface must pass an HRESULT as the function's return value. If the method is declared using the safecall calling convention, this condition is imposed automatically, with the declared return type converted to an output parameter.
- Members of a dual interface that need to return other values should specify these parameters as **var** or **out**, indicating an output parameter that returns the value of the function.

Note One way to bypass the Automation types restrictions is to implement a separate *IDispatch* interface and a custom interface. By doing so, you can use the full range of possible argument types. This means that COM clients have the option of using the custom interface, which Automation controllers can still access. In this case, though, you must implement the marshaling code manually.

Custom marshaling

Typically, you use automatic marshaling in out-of-process and remote servers because it is easier—COM does the work for you. However, you may decide to provide custom marshaling if you think you can improve marshaling performance. When implementing your own custom marshaling, you must support the *IMarshal* interface. For more information, on this approach, see the Microsoft documentation.

Registering a COM object

You can register your server object as an in-process or an out-of-process server. For more information on the server types, see"In-process, out-of-process, and remote servers" on page 33-6.

Note Before you remove a COM object from your system, you should unregister it.

Registering an in-process server

To register an in-process server (DLL or OCX),

- Choose Run | Register ActiveX Server.

To unregister an in-process server,

- Choose Run | Unregister ActiveX Server.

Registering an out-of-process server

To register an out-of-process server,

- Run the server with the **/regserver** command-line option.

  You can set command-line options with the Run | Parameters dialog box.

  You can also register the server by running it.

To unregister an out-of-process server,

- Run the server with the **/unregserver** command-line option.

As an alternative, you can use the **tregsvr** command from the command line or run the regsvr32.exe from the operating system.

Note If the COM server is intended for use under COM+, you should install it in a COM+ application rather than register it. (Installing the object in a COM+ application automatically takes care of registration.) For information on how to install an object in a COM+ application, see "Installing transactional objects" on page 39-22.

Testing and debugging the application

To test and debug your COM server application,

1. Turn on debugging information using the Compiler tab on the Project | Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools | Debugger Options dialog.
2. For an in-process server, choose Run | Parameters, type the name of the Automation controller in the Host Application box, and choose OK.
3. Choose Run | Run.
4. Set breakpoints in the Automation server.
5. Use the Automation controller to interact with the Automation server.

The Automation server pauses when the breakpoints are reached.

Note As an alternate approach, if you are also writing the Automation controller, you can debug into an in-process server by enabling COM cross-process support. Use the General page of the Tools | Debugger Options dialog to enable cross-process support.

Chapter

37

Creating an Active Server Page

If you are using the Microsoft Internet Information Server (IIS) environment to serve your Web pages, you can use Active Server Pages (ASP) to create dynamic Web-based client-server applications. Active Server Pages let you write a script that gets called every time the server loads the Web page. This script can, in turn, call on Automation objects to obtain information that it includes in a generated HTML page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and use this control to access data that gets updated every time the server loads the Web page.

On the client side, the ASP acts like a standard HTML document and can be viewed by users on any platform using any Web Browser.

ASP applications are analogous to applications you write using Delphi's Web broker technology. For more information about the Web broker technology, see Chapter 27, "Creating Internet applications". ASP differs, however, in the way it separates the UI design from the implementation of business rules or complex application logic.

- The UI design is managed by the Active Server Page. This is essentially an HTML document, but it can include embedded script that calls on Active Server objects to supply it with content that reflects your business rules or application logic.
- The application logic is encapsulated by Active Server objects that expose simple methods to the Active Server Page, supplying it with the content it needs.

Note Although ASP provides the advantage of separating UI design from application logic, its performance is limited in scale. For Web sites that respond to extremely large numbers of clients, an approach based on the Web broker technology is recommended instead.

The script in your Active Server Pages and the Automation objects you embed in an active server page can make use of the ASP intrinsics (built-in objects that provide information about the current application, HTTP messages from the browser, and so on).

This chapter shows how to create an Active Server Object using the Delphi Active Server Object wizard. This special Automation control can then be called by an Active Server Page and supply it with content.

Here are the steps for creating an Active Server Object:

- Create an Active Server Object for the application.
- Define the Active Server Object's interface.
- Register the Active Server Object.
- Test and debug the application.

Creating an Active Server Object

An Active Server Object is an Automation object that has access to information about the entire ASP application and the HTTP messages it uses to communicate with browsers. It descends from *TASPObject* or *TASPMTSObject* (which is in turn a descendant of *TAutoObject*), and supports Automation protocols, exposing itself for other applications (or the script in the Active Server page) to use. You create an Active Server Object using the Active Server Object wizard.

Your Active Server Object project can be either an executable (exe) or library (dll), depending on your needs. However, you should be aware of the drawbacks of using an out-of-process server. These drawbacks are discussed in "Creating ASPs for in-process or out-of-process servers" on page 37-7.

To display the Active Server Object wizard:

1. Choose File | New | Other.
2. Select the tab labeled, ActiveX.
3. Double-click the Active Server Object icon.

In the wizard, give your new Active Server Object a name, and specify the instancing and threading models you want to support. These details influence the way your object can be called. You must write the implementation so that it adheres to the model (for example, avoiding thread conflicts). The instancing and threading models involve the same choices that you make for other COM objects. For details, see "COM object instancing types" on page 36-5 and "Choosing a threading model" on page 36-6.

The thing that makes an Active Server Object unique is its ability to access information about the ASP application and the HTTP messages that pass between the Active Server page and client Web browsers. This information is accessed using the ASP intrinsics. In the wizard, you can specify how your object accesses these by setting the Active Server Type:

- If you are working with IIS 3 or IIS 4, you use Page Level Event Methods. Under this model, your object implements the methods, *OnStartPage* and *OnEndPage*, which are called when the Active Server page loads and unloads. When your object is loaded, it automatically obtains an *IScriptingContext* interface, which it uses to access the ASP intrinsics. These interfaces are, in turn, surfaced as properties inherited from the base class (*TASPObject*).

- If you are working with IIS5 or later, you use the Object Context type. Under this model, your object fetches an *IObjectContext* interface, which it uses to access the ASP intrinsics. Again, these interfaces are surfaced as properties in the inherited base class (*TASPMTSObject*). One advantage of this latter approach is that your object has access to all of the other services available through *IObjectContext*. To access the *IObjectContext* interface, simply call *GetObjectContext* (defined in the mtx unit) as follows:

  ```
  ObjectContext := GetObjectContext;
  ```

  For more information about the services available through *IObjectContext*, see Chapter 39, “Creating MTS or COM+ objects”.

You can tell the wizard to generate a simple ASP page to host your new Active Server Object. The generated page provides a minimal script (written in VBScript) that creates your Active Server Object based on its ProgID, and indicates where you can call its methods. This script calls **Server.CreateObject** to launch your Active Server Object.

Note Although the generated test script uses VBScript, Active Server Pages also can be written using Jscript.

When you exit the wizard, a new unit is added to the current project that contains the definition for the Active Server Object. In addition, the wizard adds a type library project and opens the Type Library editor. Now you can expose the properties and methods of the interface through the type library as described in “Defining a COM object’s interface” on page 36-9. As you write the implementation of your object’s properties and methods, you can take advantage of the ASP intrinsics (described below) to obtain information about the ASP application and the HTTP messages it uses to communicate with browsers.

The Active Server Object, like any other Automation object, implements a **dual interface**, which supports both early (compile-time) binding through the VTable and late (runtime) binding through the *IDispatch* interface. For more information on dual interfaces, see “Dual interfaces” on page 36-13.

Using the ASP intrinsics

The ASP intrinsics are a set of COM objects supplied by ASP to the objects running in an Active Server Page. They let your Active Server Object access information that reflects the messages passing between your application and the Web browser, as well as a place to store information that is shared among Active Server Objects that belong to the same ASP application.

To make these objects easy to access, the base class for your Active Server Object surfaces them as properties. For a complete understanding of these objects, see the Microsoft documentation. However, the following topics provide a brief overview.

Application

The Application object is accessed through an *IApplicationObject* interface. It represents the entire ASP application, which is defined as the set of all .asp files in a virtual directory and its subdirectories. The Application object can be shared by multiple clients, so it includes locking support that you should use to prevent thread conflicts.

IApplicationObject includes the following:

Table 37.1 IApplicationObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the application using script commands. This interface has two methods, *Remove* and *RemoveAll*, that you can use to delete one or all objects from the list.
StaticObjects property	Lists all the objects that were added to the application with the <OBJECT> tag.
Lock method	Prevents other clients from locking the Application object until you call Unlock. All clients should call Lock before accessing shared memory (such as the properties).
Unlock method	Releases the lock that was set using the Lock method.
Application_OnEnd event	Occurs when the application quits, after the Session_OnEnd event. The only intrinsics available are Application and Server. The event handler must be written in VBScript or JScript.
Application_OnStart event	Occurs before the new session is created (before Session_OnStart). The only intrinsics available are Application and Server. The event handler must be written in VBScript or JScript.

Request

The Request object is accessed through an *IRequest* interface. It provides information about the HTTP request message that caused the Active Server Page to be opened.

IRequest includes the following:

Table 37.2 IRequest interface members

Property, Method, or Event	Meaning
ClientCertificate property	Indicates the values of all fields in the client certificate that is sent with the HTTP message.
Cookies property	Indicates the values of all Cookie headers on the HTTP message.
Form property	Indicates the values of form elements in the HTTP body. These can be accessed by name.
QueryString property	Indicates the values of all variables in the query string from the HTTP header.
ServerVariables property	Indicates the values of various environment variables. These variables represent most of the common HTTP header variables.
TotalBytes property	Indicates the number of bytes in the request body. This is an upper limit on the number of bytes returned by the BinaryRead method.
BinaryRead method	Retrieves the content of a Post message. Call the method, specifying the maximum number of bytes to read. The resulting content is returns as a Variant array of bytes. After calling BinaryRead, you can't use the Form property.

Response

The Request object is accessed through an *IResponse* interface. It lets you specify information about the HTTP response message that is returned to the client browser.

IResponse includes the following:

Table 37.3 IResponse interface members

Property, Method, or Event	Meaning
Cookies property	Determines the values of all Cookie headers on the HTTP message.
Buffer property	Indicates whether page output is buffered When page output is buffered, the server does not send a response to the client until all of the server scripts on the current page are processed.
CacheControl property	Determines whether proxy servers can cache the output in the response.
Charset property	Adds the name of the character set to the content type header.
ContentType property	Specifies the HTTP content type of the response message's body.
Expires property	Specifies how long the response can be cached by a browser before it expires.
ExpiresAbsolute property	Specifies the date and time when the response expires.
IsClientConnected property	Indicates whether the client has disconnected from the server.
Pics property	Set the value for the pics-label field of the response header.
Status property	Indicates the status of the response. This is the value of an HTTP status header.
AddHeader method	Adds an HTTP header with a specified name and value.
AppendToLog method	Adds a string to the end of the Web server log entry for this request.
BinaryWrite method	Writes raw (uninterpreted) information to the body of the response message.
Clear method	Erases any buffered HTML output.
End method	Stops processing the .asp file and returns the current result.
Flush method	Sends any buffered output immediately.
Redirect method	Sends a redirect response message, redirecting the client browser to a different URL.
Write method	Writes a variable to the current HTTP output as a string.

Session

The Session object is accessed through the *ISessionObject* interface. It allows you to store variables that persist for the duration of a client's interaction with the ASP application. That is, these variables are not freed when the client moves from page to page within the ASP application, but only when the client exits the application altogether.

ISessionObject includes the following:

Table 37.4 ISessionObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the session using the <OBJECT> tag. You can access any variable in the list by name, or call the Contents object's *Remove* or *RemoveAll* method to delete values.
StaticObjects property	Lists all the objects that were added to the session with the <OBJECT> tag.
CodePage property	Specifies the code page to use for symbol mapping. Different locales may use different code pages.
LCID property	Specifies the locale identifier to use for interpreting string content.
SessionID property	Indicates the session identifier for the current client.
Timeout property	Specifies the time, in minutes, that the session persists without a request (or refresh) from the client until the application terminates.
Abandon method	Destroys the session and releases its resources.
Session_OnEnd event	Occurs when the session is abandoned or times out. The only intrinsics available are Application, Server, and Session. The event handler must be written in VBScript or JScript.
Session_OnStart event	Occurs when the server creates a new session is created (after Application_OnStart but before running the script on the Active Server Page). All intrinsics are available. The event handler must be written in VBScript or JScript.

Server

The Server object is accessed through an *IServer* interface. It provides various utilities for writing your ASP application.

IServer includes the following:

Table 37.5 IServer interface members

Property, Method, or Event	Meaning
ScriptTimeout property	Same as the Timeout property on the Session object.
CreateObject method	Instantiates a specified Active Server Object.
Execute method	Executes the script in a specified .asp file.
GetLastError method	Returns an ASPError object that describes the error condition.
HTMLEncode method	Encodes a string for use in an HTML header, replacing reserved characters by the appropriate symbolic constants.
MapPath method	Maps a specified virtual path (an absolute path on the current server or a path relative to the current page) into a physical path.
Transfer method	Sends all of the current state information to another Active Server Page for processing.
URLEncode method	Applies URL encoding rules, including escape characters, to a specified string

Creating ASPs for in-process or out-of-process servers

You can use **Server.CreateObject** in an ASP page to launch either an in-process or out-of-process server, depending on your requirements. However, launching in-process servers is more common.

Unlike most in-process servers, an Active Server Object in an in-process server does not run in the client's process space. Instead, it runs in the IIS process space. This means that the client does not need to download your application (as, for example, it does when you use ActiveX objects). In-process component DLLs are faster and more secure than out-of-process servers, so they are better suited for server-side use.

Because out-of-process servers are less secure, it is common for IIS to be configured to *not* allow out-of-process executables. In this case, creating an out-of-process server for your Active Server Object would result in an error similar to the following:

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

Also, out-of-process components often create individual server processes for each object instance, so they are slower than CGI applications. They do not scale as well as component DLLs.

If performance and scalability are priorities for your site, in-process servers are highly recommended. However, Intranet sites that receive moderate to low traffic may use an out-of-process component without adversely affecting the site's overall performance.

For general information on in-process and out-of-process servers, see, "In-process, out-of-process, and remote servers," on page 33-6.

Registering an Active Server Object

You can register the Active Server Page as an in-process or an out-of-process server. However, in-process servers are more common.

Note When you want to remove the Active Server Page object from your system, you should first unregister it, removing its entries from the Windows registry.

Registering an in-process server

To register an in-process server (DLL or OCX),

- Choose Run | Register ActiveX Server.

To unregister an in-process server,

- Choose Run | Unregister ActiveX Server.

Registering an out-of-process server

To register an out-of-process server,

- Run the server with the /regserver command-line option. (You can set command-line options with the Run|Parameters dialog box.)

  You can also register the server by running it.

To unregister an out-of-process server,

- Run the server with the /unregserver command-line option.

Testing and debugging the Active Server Page application

Debugging any in-process server such as an Active Server Object is much like debugging a DLL. You choose a host application that loads the DLL, and debug as usual. To test and debug an Active Server Object,

1. Turn on debugging information using the Compiler tab on the Project|Options dialog box, if necessary. Also, turn on Integrated Debugging in the Tools|Debugger Options dialog.
2. Choose Run|Parameters, type the name of your Web Server in the Host Application box, and choose OK.
3. Choose Run|Run.
4. Set breakpoints in the Active Server Object implementation.
5. Use the Web browser to interact with the Active Server Page.

The debugger pauses when the breakpoints are reached.

Chapter

38

Creating an ActiveX control

An ActiveX control is a software component that integrates into and extends the functionality of any host application that supports ActiveX controls, such as C++Builder, Delphi, Visual Basic, Internet Explorer, and (given a plug-in) Netscape Navigator. ActiveX controls implement a particular set of interfaces that allow this integration.

For example, Delphi comes with several ActiveX controls, including charting, spreadsheet, and graphics controls. You can add these controls to the component palette in the IDE, and then use them like any standard VCL component, dropping them on forms and setting their properties using the Object Inspector.

An ActiveX control can also be deployed on the Web, allowing it to be referenced in HTML documents and viewed with ActiveX-enabled Web browsers.

Delphi provides wizards that let you create two types of ActiveX controls:

- **ActiveX controls that wrap VCL classes.** By wrapping a VCL class, you can convert existing components into ActiveX controls or create new ones, test them out locally, and then convert them into ActiveX controls. ActiveX controls are typically intended to be embedded in a larger host application.
- **Active forms.** Active forms let you use the form designer to create a more elaborate control that acts like a dialog or like a complete application. You develop the Active form in much the same way that you develop a typical Delphi application. Active Forms are typically intended for deployment on the Web.

This chapter provides an overview of how to create an ActiveX control in the Delphi environment. It is not intended to provide complete implementation details of writing ActiveX control without using a wizard. For that information, refer to your Microsoft Developer's Network (MSDN) documentation or search the Microsoft Web site for ActiveX information.

Overview of ActiveX control creation

Creating ActiveX controls using Delphi is very similar to creating ordinary controls or forms. This differs markedly from creating other COM objects, where you first define the object's interface and then complete the implementation. To create ActiveX controls (other than Active Forms), you reverse this process, starting with the implementation of a VCL control, and then generating the interface and type library once the control is written. When creating Active Forms, the interface and type library are created at the same time as your form, and then you use the form designer to implement the form.

The completed ActiveX control consists of a VCL control that provides the underlying implementation, a COM object that wraps the VCL control, and a type library that lists the COM object's properties, methods, and events.

To create a new ActiveX control (other than an Active Form), perform the following steps:

1. Design and create the custom VCL control that forms the basis of your ActiveX control.
2. Use the ActiveX control wizard to create an ActiveX control from the VCL control you created in step 1.
3. Use the ActiveX property page wizard to create one or more property pages for the control (optional).
4. Associate the property page with the ActiveX control (optional).
5. Register the control.
6. Test the control with all potential target applications.
7. Deploy the ActiveX control on the Web. (optional)

To create a new Active Form, perform the following steps:

1. Use the ActiveForm wizard to create an Active Form, which appears as a blank form in the IDE, and an associated ActiveX wrapper for that form.
2. Use the form designer to add components to your Active Form and implement its behavior in the same way you create and implement an ordinary form using the form designer.
3. Follow steps 3-7 above to give your Active Form a property page, register it, and deploy it on the Web.

Elements of an ActiveX control

An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, a corresponding COM object wrapper that exposes properties, methods, and events, and one or more associated type libraries.

VCL control

The underlying implementation of an ActiveX control in Delphi is a VCL control. When you create an ActiveX control, you must first design or choose the VCL control from which you will make your ActiveX control.

The underlying VCL control must be a descendant of *TWinControl*, because it must have a window that can be parented by the host application. When you create an Active form, this object is a descendant of *TActiveForm*.

Note The ActiveX control wizard lists the available *TWinControl* descendants from which you can choose to make an ActiveX control. This list does not include all *TWinControl* descendants, however. Some controls, such as *THeaderControl*, are registered as incompatible with ActiveX (using the *RegisterNonActiveX* procedure) and do not appear in the list.

ActiveX wrapper

The actual COM object is an ActiveX wrapper object for the VCL control. For Active forms, this class is always *TActiveFormControl*. For other ActiveX controls, it has a name of the form *TVCLClass*X, where *TVCLClass* is the name of the VCL control class. Thus, for example, the ActiveX wrapper for *TButton* would be named *TButtonX*.

The wrapper class is a descendant of *TActiveXControl*, which provides support for the ActiveX interfaces. The ActiveX wrapper inherits this support, which allows it to forward Windows messages to the VCL control and parent its window in the host application.

The ActiveX wrapper exposes the VCL control's properties and methods to clients via its default interface. The wizard automatically implements most of the wrapper class's properties and methods, delegating method calls to the underlying VCL control. The wizard also provides the wrapper class with methods that fire the VCL control's events on clients and assigns these methods as event handlers on the VCL control.

Type library

The ActiveX control wizards automatically generate a type library that contains the type definitions for the wrapper class, its default interface, and any type definitions that these require. This type information provides a way for your control to advertise its services to host applications. You can view and edit this information using the Type Library editor. Although this information is stored in a separate, binary type library file (.TLB extension), it is also automatically compiled into the ActiveX control DLL as a resource.

Property page

You can optionally give your ActiveX control a property page. The property page allows the user of a host (client) application to view and edit your control's properties. You can group several properties on a page, or use a page to provide a dialog-like interface for a property. For information on how to create property pages, see "Creating a property page for an ActiveX control" on page 38-11.

Designing an ActiveX control

When designing an ActiveX control, you start by creating a custom VCL control. This forms the basis of your ActiveX control. For information on creating custom controls, see Part V, “Creating custom components.”

When designing the VCL control, keep in mind that it will be embedded in another application; this control is not an application in itself. For this reason, you probably do not want to use elaborate dialog boxes or other major user-interface components. Your goal is typically to make a simple control that works inside of, and follows the rules of the main application.

In addition, you should make sure that the types for all properties and methods you want your object to expose to clients are Automation-compatible, because the ActiveX control’s interface must support *IDispatch*. The wizard does not add any methods to the wrapper class’s interface that have parameters that are not Automation-compatible. For a list of Automation-compatible types, see “Valid types” on page 34-11.

The wizards implement all the necessary ActiveX interfaces required using the COM wrapper class. They also surface all Automation-compatible properties, methods, and events through the wrapper class’s default interface. Once the wizard has generated the COM wrapper class and its interface, you can use the Type Library editor to modify the default interface or augment the wrapper class by implementing additional interfaces.

Generating an ActiveX control from a VCL control

To generate an ActiveX control from a VCL control, use the ActiveX Control wizard. The properties, methods, and events of the VCL control become the properties, methods, and events of the ActiveX control.

Before using the ActiveX control wizard, you must decide what VCL control will provide the underlying implementation of the generated ActiveX control.

To bring up the ActiveX control wizard,

1. Choose File | New | Other to open the New Items dialog box.
2. Select the tab labeled ActiveX.
3. Double-click the ActiveX Control icon.

In the wizard, select the name of the VCL control that will be wrapped by the new ActiveX control. The dialog lists all available controls, which are descendants of *TWinControl* that are not registered as incompatible with ActiveX using the *RegisterNonActiveX* procedure.

Tip If you do not see the control you want in the drop-down list, check whether you have installed it in the IDE or added its unit to your project.

Once you have selected a VCL control, the wizard automatically generates a name for the CoClass, the implementation unit for the ActiveX wrapper, and the ActiveX library project. (If you currently have an ActiveX library project open, and it does not contain a COM+ event object, the current project is automatically used.) You can change any of these in the wizard (unless you have an ActiveX library project already open, in which case the project name is not editable).

The wizard always specifies Apartment as the threading model. This is not a problem if your ActiveX project usually contains only a single control. However, if you add additional objects to your project, you are responsible for providing thread support.

The wizard also lets you configure various options on your ActiveX control:

- **Enabling licensing:** You can make your control licensed to ensure that users of the control can't open it either for design purposes or at runtime unless they have a license key for the control.
- **Including Version information:** You can include version information, such as a copyright or a file description, in the ActiveX control. This information can be viewed in a browser. Some host clients, such as Visual Basic 4.0, require Version information or they will not host the ActiveX control. Specify version information by choosing Project | Options and selecting the Version Info page.
- **Including an About box:** You can tell the wizard to generate a separate form that implements an About box for your control. Users of the host application can display this About box in a development environment. By default, the About box includes the name of the ActiveX control, an image, copyright information, and an OK button. You can modify this default form, which the wizard adds to your project.

When you exit the wizard, it generates the following:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don’t change this file.
- A type library, which defines and CoClass for your control, the interface it exposes to clients, and any type definitions that these require. For more information about the type library, refer to Chapter 34, “Working with type libraries.”
- An ActiveX implementation unit, which defines and implements the ActiveX control, a descendant of *TActiveXControl*. This ActiveX control is a fully-functioning implementation that requires no additional work on your part. However, you can modify this class if you want to customize the properties, methods, and events that the ActiveX control exposes to clients.
- An About box form and unit if you requested them.
- A .LIC file if you enabled licensing.

Generating an ActiveX control based on a VCL form

Unlike other ActiveX controls, Active Forms are not first designed and then wrapped by an ActiveX wrapper class. Instead, the ActiveForm wizard generates a blank form that you design later when the wizard leaves you in the Form Designer.

When an ActiveForm is deployed on the Web, Delphi creates an HTML page to contain the reference to the ActiveForm and specify its location on the page. The ActiveForm can then displayed and run from a Web browser. Inside the browser, the form behaves just like a stand-alone Delphi form. The form can contain any VCL components or ActiveX controls, including custom-built VCL controls.

To start the ActiveForm wizard,

1 Choose File | New | Other to open the New Items dialog box.

2 Select the tab labeled ActiveX.

3 Double-click the ActiveForm icon.

The Active Form wizard looks just like the ActiveX control wizard, except that you can't specify the name of the VCL class to wrap. This is because Active forms are always based on *TActiveForm*.

As in the ActiveX control wizard, you can change the default names for the CoClass, implementation unit, and ActiveX library project. Similarly, this wizard lets you indicate whether you want your Active Form to require a license, whether it should include version information, and whether you want an About box form.

When you exit the wizard, it generates the following:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A type library, which defines and CoClass for your control, the interface it exposes to clients, and any type definitions that these require. For more information about the type library, refer to Chapter 34, "Working with type libraries."
- A form that descends from *TActiveForm*. This form appears in the form designer, where you can use it to visually design the Active Form that appears to clients. Its implementation appears in the generated implementation unit. In the initialization section of the implementation unit, a class factory is created, setting up *TActiveFormControl* as the ActiveX wrapper for this form.
- An About box form and unit if you requested them.
- A .LIC file if you enabled licensing.

At this point, you can add controls and design the form as you like.

After you have designed and compiled the ActiveForm project into an ActiveX library (which has the OCX extension), you can deploy the project to your Web server and Delphi creates a test HTML page with a reference to the ActiveForm.

Licensing ActiveX controls

Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically for controls created at runtime.

To provide design-time licenses, the ActiveX wizard creates a key for the control, which it stores in a file with the same name as the project with the LIC extension. This .LIC file is added to the project. The user of the control must have a copy of the .LIC file to open the control in a development environment. Each control in the project that has Make Control Licensed checked has a separate key entry in the LIC file.

To support runtime licenses, the wrapper class implements two methods, *GetLicenseString* and *GetLicenseFilename*. These return the license string for the control and the name of the .LIC file, respectively. When a host application tries to create the ActiveX control, the class factory for the control calls these methods and compares the string returned by *GetLicenseString* with the string stored in the .LIC file.

Runtime licenses for the Internet Explorer require an extra level of indirection because users can view HTML source code for any Web page, and because an ActiveX control is copied to the user's computer before it is displayed. To create runtime licenses for controls used in Internet Explorer, you must first generate a license package file (LPK file) and embed this file in the HTML page that contains the control. The LPK file is essentially an array of ActiveX control CLSIDs and license keys.

Note To generate the LPK file, use the utility, LPK_TOOL.EXE, which you can download from the Microsoft Web site (www.microsoft.com).

To embed the LPK file in a Web page, use the HTML objects, <OBJECT> and <PARAM> as follows:

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
    <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

The CLSID identifies the object as a license package and PARAM specifies the relative location of the license package file with respect to the HTML page.

When Internet Explorer tries to display the Web page containing the control, it parses the LPK file, extracts the license key, and if the license key matches the control's license (returned by *GetLicenseString*), it renders the control on the page. If more than one LPK is included in a Web page, Internet Explorer ignores all but the first.

For more information, look for Licensing ActiveX Controls on the Microsoft Web site.

Customizing the ActiveX control's interface

The ActiveX Control and ActiveForm wizards generate a default interface for the ActiveX wrapper class. This default interface simply exposes the properties, methods, and events of the original VCL control or form, with the following exceptions:

- Data-aware properties do not appear. Because ActiveX controls have a different mechanism for making controls data-aware than VCL controls, the wizards do not convert properties related to data. See "Enabling simple data binding with the type library" on page 38-10 for information on how to make your ActiveX control data-aware.
- Any property, method, or event that type that is not Automation-compatible does not appear. You may want to add these to the ActiveX control's interface after the wizard has finished.

You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library. You can use the Type Library editor as described in Chapter 34, "Working with type libraries."Remember that when you add events, they should be added to the Events interface, not the ActiveX control's default interface.

Note You can add unpublished properties to your ActiveX control's interface. Such properties can be set at runtime and will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value of a property at design time, the changes are not reflected when the control is run. If the source is a VCL object and the property is not already published, you can make properties persistent by creating a descendant of the VCL object and publishing the property in the descendant.

You may also choose not to expose all of the VCL control's properties, methods, and events to host applications. You can use the Type Library editor to remove these from the interfaces that the wizard generated. When you remove properties and methods from an interface using the Type Library editor, the Type Library editor does not remove them from the corresponding implementation class. Edit the ActiveX wrapper class in the implementation unit to remove these after you have changed the interface in the Type Library editor.

Warning Any changes you make to the type library will be lost if you regenerate the ActiveX control from the original VCL control or form.

Tip It is a good idea to check the methods that the wizard adds to your ActiveX wrapper class. Not only does this give you a chance to note where the wizard omitted any data-aware properties or methods that were not Automation-compatible, it also lets you detect methods for which the wizard could not generate an implementation. Such methods appear with a comment in the implementation that indicates the problem.

Adding additional properties, methods, and events

You can add additional properties, methods, and events to the control using the type library editor. The declaration is automatically added to the control's implementation unit, type library (TLB) file, and type library unit. The specifics of what Delphi supplies depends on whether you have added a property or method or whether you have added an event.

Adding properties and methods

The ActiveX wrapper class implements properties in its interface using read and write access methods. That is, the wrapper class has COM properties, which appear on an interface as getter and/or setter methods. Unlike VCL properties, you do not see a "property" declaration on the interface for COM properties. Rather, you see methods that are flagged as property access methods. When you add a property to the ActiveX control's default interface, the wrapper class definition (which appears in the _TLB unit that is updated by the Type Library editor) gains one or two new methods (a getter and/or setter) that you must implement, just as when you add a method to the interface, the wrapper class gains a corresponding method for you to implement. Thus, adding properties to the wrapper class's interface is essentially the same as adding methods: the wrapper class definition gains new skeletal method implementations for you to complete.

Note For details on what appears in the generated _TLB unit, see "Code generated when you import type library information" on page 35-5.

For example, consider a *Caption* property, of type *TCaption* in the underlying VCL object. To Add this property to the object's interface, you enter the following when you add a property to the interface via the type library editor:

```
property Caption: TCaption read Get_Caption write Set_Caption;
```

Delphi adds the following declarations to the wrapper class:

```
function Get_Caption: WideString; safecall;
procedure Set_Caption(const Value: WideString); safecall;
```

In addition, it adds skeletal method implementations for you to complete:

```
function TButtonX.Get_Caption: WideString;
begin
end;

procedure TButtonX.Set_Caption(Value: WideString);
begin
end;
```

Typically, you can implement these methods by simply delegating to the associated VCL control, which can be accessed using the *FDelphiControl* member of the wrapper class:

```
function TButtonX.Get_Caption: WideString;
begin
  Result := WideString(FDelphiControl.Caption);
end;

procedure TButtonX.Set_Caption(const Value: WideString);
begin
  FDelphiControl.Caption := TCaption(Value);
end;
```

In some cases, you may need to add code to convert the COM data types to native Object Pascal types. The preceding example manages this with typecasting.

Note Because the Automation interface methods are declared **safecall**, you do not have to implement COM exception code for these methods—the Delphi compiler handles this for you by generating code around the body of **safecall** methods to catch Delphi exceptions and to convert them into COM error info structures and return codes.

Adding events

The ActiveX control can fire events to its container in the same way that an automation object fires events to clients. This mechanism is described in "Exposing events to clients" on page 36-10.

If the VCL control you are using as the basis of your ActiveX control has any published events, the wizards automatically add the necessary support for managing a list of client event sinks to your ActiveX wrapper class and define the outgoing dispinterface that clients must implement to respond to events.

You add events to this outgoing dispinterface. To add an event in the type library editor, select the event interface and click on the method icon. Then manually add the list of parameters you want include using the parameter page.

Next, you must declare a method in your wrapper class that is of the same type as the event handler for the event in the underlying VCL control. This is not generated automatically, because Delphi does not know which event handler you are using:

```
  procedure KeyPressEvent(Sender: TObject; var Key: Char);
```

Implement this method to use the host application's event sink, which is stored in the wrapper class's *FEvents* member:

```
procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
    TempKey := Smallint(Key); {cast to an OleAutomation compatible type }
    if FEvents <> nil then
      FEvents.OnKeyPress(TempKey)
    Key := Char(TempKey);
  end;
```

Note When firing events in an ActiveX control, you do not need to iterate through a list of event sinks because the control only has a single host application. This is simpler than the process for most Automation servers.

Finally, you must assign this event handler to the underlying VCL control, so that it is called when the event occurs. You make this assignment in the *InitializeControl* method:

```
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
```

Enabling simple data binding with the type library

With simple data binding, you can bind a property of your ActiveX control to a field in a database. To do this, the ActiveX control must communicate with its host application about what value represents field data and when it changes. You enable this communication by setting the property's binding flags using the Type Library editor.

By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies its container (the client host application) that the value has changed and requests that the database record be updated. The container interacts with the database and then notifies the control whether it succeeded or failed to update the record.

Note The container application that hosts your ActiveX control is responsible for connecting the data-aware properties you enable in the type library to the database. See "Using data-aware ActiveX controls" on page 35-8 for information on how to write such a container using Delphi.

Use the type library to enable simple data binding,

1. On the toolbar, click the property that you want to bind.
2. Choose the flags page.
3. Select the following binding attributes:

Binding attribute	Description
Bindable	Indicates that the property supports data binding. If marked bindable, the property notifies its container when the property value has changed.
Request Edit	Indicates that the property supports the OnRequestEdit notification. This allows the control to ask the container if its value can be edited by the user.
Display Bindable	Indicates that the container can show users that this property is bindable.
Default Bindable	Indicates the single, bindable property that best represents the object. Properties that have the default bind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Immediate Bindable	Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified. The bindable and request edit attribute bits need to be set for this new bit to have an effect.

4. Click the Refresh button on the toolbar to update the type library.

To test a data-binding control, you must register it first.

For example, to convert a *TEdit* control into a data-bound ActiveX control, create the ActiveX control from a *TEdit* and then change the Text property flags to Bindable, Display Bindable, Default Bindable, and Immediate Bindable. After the control is registered and imported, it can be used to display data.

Creating a property page for an ActiveX control

A property page is a dialog box similar to the Delphi Object Inspector in which users can change the properties of an ActiveX control. A property page dialog allows you to group many properties for a control together to be edited at once. Or, you can provide a dialog box for more complex properties.

Typically, users access the property page by right-clicking the ActiveX control and choosing Properties.

The process of creating a property page is similar to creating a form, you

1. Create a new property page.
2. Add controls to the property page.
3. Associate the controls on the property page with the properties of an ActiveX control.
4. Connect the property page to the ActiveX control.

Note When adding properties to an ActiveX control or ActiveForm, you must publish the properties that you want to persist. If they are not published in the underlying VCL control, you must make a custom descendant of the VCL control that redeclares the properties as published and then use the ActiveX control wizard to create an ActiveX control from the descendant class.

Creating a new property page

You use the Property Page wizard to create a new property page.

To create a new property page,

1. Choose File | New | Other.
2. Select the ActiveX tab.
3. Double-click the Property Page icon.

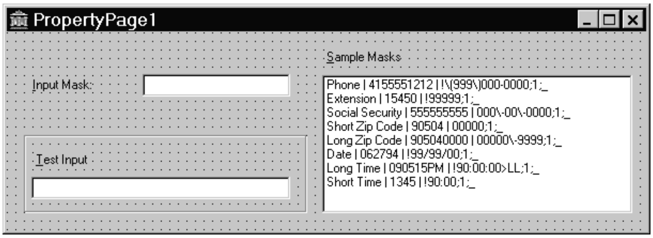
The wizard creates a new form and implementation unit for the property page. The form is a descendant of *TPropertyPage*, which lets you associate the form with the ActiveX control whose properties it edits.

Adding controls to a property page

You must add a control to the property page for each property of the ActiveX control that you want the user to access.

For example, the following illustration shows a property page for setting the MaskEdit property of an ActiveX control.

Figure 38.1 Mask Edit property page in design mode



```
Phone | 4155551212 | !\(999\)000-0000;1;_
Extension | 15450 | !99999;1;_
Social Security | 555555555 | 000\-00\-0000;1;_
Short Zip Code | 90504 | 00000;1;_
Long Zip Code | 905040000 | 00000\-9999;1;_
Date | 062794 | !99/99/00;1;_
Long Time | 090515PM | !90:00:00>LL;1;_
Short Time | 1345 | !90:00;1;_
```

The list box allows the user to select from a list of sample masks. The edit controls allow the user to test the mask before applying it to the ActiveX control. You add controls to the property page the same as you would to a form.

Associating property page controls with ActiveX control properties

After adding the controls you need to the property page, you must associate each control with its corresponding property. You make this association by adding code to the property page's *UpdatePropertyPage* and *UpdateObject* methods.

Updating the property page

Add code to the *UpdatePropertyPage* method to update the control on the property page when the properties of the ActiveX control change. You must add code to the *UpdatePropertyPage* method to update the property page with the current values of the ActiveX control's properties.

You can access the ActiveX control using the property page's *OleObject* property, which is an *OleVariant* that contains the ActiveX control's interface.

For example, the following code updates the property page's edit control (InputMask) with the current value of the ActiveX control's *EditMask* property:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
   { Update your controls from OleObject }
   InputMask.Text := OleObject.EditMask;
end;
```

Note It is also possible to write a property page that represents more than one ActiveX control. In this case, you don't use the *OleObject* property. Instead, you must iterate through a list of interfaces that is maintained by the *OleObjects* property.

Updating the object

Add code to the *UpdateObject* method to update the property when the user changes the controls on the property page. You must add code to the *UpdateObject* method in order to set the properties of the ActiveX control to their new values.

Once again you use the *OleObject* property to access the ActiveX control.

For example, the following code sets the *EditMask* property of the ActiveX control using the value in the property page's edit box control (InputMask):

```
procedure TPropertyPage1.UpdateObject;
begin
   {Update OleObject from your control }
   OleObject.EditMask := InputMask.Text;
end;
```

Connecting a property page to an ActiveX control

To connect a property page to an ActiveX control,

1 Add *DefinePropertyPage* with the GUID constant of the property page as the parameter to the *DefinePropertyPages* method implementation in the control's implementation for the unit. For example,

```
procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

The GUID constant, Class_PropertyPage1, of the property page can be found in the property pages unit.

The GUID is defined in the property page's implementation unit; it is generated automatically by the Property Page wizard.

2 Add the property page unit to the **uses** clause of the controls implementation unit.

Registering an ActiveX control

After you have created your ActiveX control, you must register it so that other applications can find and use it.

To register an ActiveX control:

- Choose Run | Register ActiveX Server.

Note Before you remove an ActiveX control from your system, you should unregister it.

To unregister an ActiveX control:

- Choose Run | Unregister ActiveX Server.

As an alternative, you can use the **tregsvr** command from the command line or run the regsvr32.exe from the operating system.

Testing an ActiveX control

To test your control, add it to a package and import it as an ActiveX control. This procedure adds the ActiveX control to the Delphi component palette. You can drop the control on a form and test as needed.

Your control should also be tested in all target applications that will use the control.

To debug the ActiveX control, select Run | Parameters and type the client name in the Host Application edit box.

The parameters then apply to the host application. Selecting Run | Run will run the host or client application and allow you to set breakpoints in the control.

Deploying an ActiveX control on the Web

Before the ActiveX controls that you create can be used by Web clients, they must be deployed on your Web server. Every time you make a change to the ActiveX control, you must recompile and redeploy it so that client applications can see the changes.

Before you can deploy your ActiveX control, you must have a Web Server that will respond to client messages.

To deploy your ActiveX control, use the following steps:

1. Select Project | Web Deployment Options.
2. On the Project page, set the Target Dir to the location of the ActiveX control DLL as a path on the Web server. This can be a local path name or a UNC path, for example, C:\INETPUB\wwwroot.
3. Set the Target URL to the location as a Uniform Resource Locators (URL) of the ActiveX control DLL (without the file name) on your Web Server, for example, http://mymachine.inprise.com/. See the documentation for your Web Server for more information on how to do this.
4. Set the HTML Dir to the location (as a path) where the HTML file that contains a reference to the ActiveX control should be placed, for example, C:\INETPUB\wwwroot. This path can be a standard path name or a UNC path.
5. Set desired Web deployment options as described in “Setting options” on page 38-16.
6. Choose OK.
7. Choose Project | Web Deploy.

   This creates a deployment code base that contains the ActiveX control in an ActiveX library (with the OCX extension). Depending on the options you specify, this deployment code base can also contain a cabinet (with the CAB extension) or information (with the INF extension).

   The ActiveX library is placed in the Target Directory you specified in step 2. The HTML file has the same name as the project file but with the HTM extension. It is created in the HTML Directory specified in step 4. The HTML file contains a URL reference to the ActiveX library at the location specified in step 3.

Note If you want to put these files on your Web server, use an external utility such as ftp.

8. Invoke your ActiveX-enabled Web browser and view the created HTML page.

When this HTML page is viewed in the Web browser, your form or control is displayed and runs as an embedded application within the browser. That is, the library runs in the same process as the browser application.

Setting options

Before deploying an ActiveX control, specify the Web deployment options that should be followed when creating the ActiveX library.

Web deployment options include settings to allow you to set the following:

- **Including additional files:** If your ActiveX control depends on any packages or other additional files, you can indicate that these should be deployed with the project. By default, these files use the same options that you specify for the entire project, but you can override these settings using the Packages or Additional files tab. When you include packages or additional files, Delphi creates a file with the .INF extension (for INFormation). This file specifies the various files that need to be downloaded and set up for the ActiveX library to run. The syntax of the INF file allows URLs pointing to packages or additional files to download.
- **CAB file compression:** A cabinet is a single file, usually with a **CAB** file extension, that stores compressed files in a file library. Cabinet compression can dramatically decrease download time (up to 70%) of a file. During installation, the browser decompresses the files stored in a cabinet and copies them to the user's system. Each file that you deploy can be CAB file compressed. You can specify that the ActiveX library use CAB file compression on the Project tab of the Web Deployment options dialog.
- **Version information:** You can specify that you want version information included with your ActiveX control. This information is set in the VersionInfo page of the Project Options dialog. Part of this information is the release number, which you can have automatically updated every time you deploy your ActiveX control. If you include additional packages or files, their Version information resources can get added to the INF file as well.

Depending on whether you include additional files and whether you use CAB file compression, the resulting ActiveX library may be an OCX file, a CAB file containing an OCX file, or an INF file. The following table summarizes the results of choosing different combinations.

Packages and/or additional files	CAB file compression	Result
No	No	An ActiveX library (OCX) file.
No	Yes	A CAB file containing an ActiveX library file.
Yes	No	An INF file, an ActiveX library file, and any additional files and packages.
Yes	Yes	An INF file, a CAB file containing an ActiveX library, and a CAB file each for any additional files and packages.

Chapter

39

Creating MTS or COM+ objects

Delphi uses the term transactional objects to refer to objects that take advantage of the transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) (for versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later). These objects are designed to work in a large, distributed environment. They are not available for use in cross-platform applications due to their dependence on Windows-specific technology.

Delphi provides a wizard that creates transactional objects so that you can take advantage of the benefits of COM+ attributes or the MTS environment. These features make creating COM clients and servers, particularly remote servers, easier to implement.

Note For database applications, Delphi also provides a Transactional Data Module. For more information, see Chapter 25, “Creating multi-tiered applications”.

Transactional objects make use of a number of low-level services, such as

- Managing system resources, including processes, threads, and database connections so that your server application can handle many simultaneous users
- Automatically initiating and controlling transactions so that your application is reliable.
- Creating, executing, and deleting server components when needed.
- Providing role-based security so that only authorized users can access your application.
- Managing events so that clients can respond to conditions that arise on the server (COM+ only).

By letting MTS or COM+ provide these underlying services, you can concentrate on developing the specifics for your particular distributed application. Which technology you choose (MTS or COM+) depends on the server on which you choose to run your application. To clients, the difference between the two (or, for that matter, the fact that the server object uses any of these services) is transparent (unless the client explicitly manipulates transactional services via a special interface).

Understanding transactional objects

Typically, transactional objects are small, and are used for discrete business functions. They can implement an application's business rules, providing views and transformations of the application state. Consider, for example, the case of a medical application. Medical records stored in various databases represent the persistent state of the application, such as a patient's health history. Transactional objects update that state to reflect such changes as new patients, test results, and X-ray files.

Transactional objects are distinguished from other COM objects in that they use a set of attributes supplied by MTS or COM+ for handling issues that arise in a distributed computing environment. Some of these attributes require the transactional object to implement the *IObjectControl* interface. *IObjectControl* defines methods that are called when the object is activated or deactivated, where you can manage resources such as database connections. It also is required for object pooling, which is described in "Object pooling" on page 39-8.

Note If you are using MTS, your transactional objects must implement *IObjectControl*. Under COM+, *IObjectControl* is not required, but is highly recommended. The Transactional Object wizard provides an object that derives from *IObjectControl*.

A client of a transactional object is called a **base client**. From a base client's perspective, a transactional object looks like any other COM object.

Under MTS, the transactional object must be built into a library (DLL), which is then installed in the MTS runtime environment (the MTS executive, mtxex.exe). That is, the server object runs in the MTS runtime process space. The MTS executive can be running in the same process as the base client, as a separate process on the same machine as the base client, or as a remote server process on a separate machine.

Under COM+, the server application need not be an in-process server. Because the various services are integrated into the COM libraries, there is no need for a separate MTS process to intercept calls to the server. Instead, COM itself (or, rather, COM+) provides the resource management, transaction support, and so on. However, the server application must still be installed, this time into a COM+ application.

The connection between the base client and the transactional object is handled by a proxy on the client and a stub on the server, just as with any out-of-process server. Connection information is maintained by the proxy. The connection between the base client and proxy remains open as long as the client requires a connection to the server, so it appears to the client that it has continued access to the server. In reality, though, the proxy may deactivate and reactivate the object, conserving resources so that other clients may use the connection. For details on activating and deactivating, see "Just-in-time activation" on page 39-4.

Requirements for a transactional object

In addition to the COM requirements, a transactional object must meet the following requirements:

- The object must have a standard class factory. This is automatically supplied by the wizard when you create the object.
- The server must expose its class object by exporting the standard *DllGetClassObject* method. Code to do this is supplied by the wizard.
- All object interfaces and CoClasses must be described by a type library, which is created automatically by the wizard. You can add methods and properties to interfaces in the type library by using the Type Library editor. The information in the type library is used by the MTS Explorer or COM+ Component Manager to extract information about the installed components at runtime.
- The server must only export interfaces that use standard COM marshaling. This is automatically supplied by the Transactional Object wizard. Delphi's support of transactional objects does not allow manual marshaling for custom interfaces. All interfaces must be implemented as dual interfaces that use COM's automatic marshaling support.
- The server must export the *DllRegisterServer* function and perform self-registration of its CLSID, ProgID, interfaces, and type library in this routine. This is provided by the Transactional Object wizard.

When using MTS rather than COM+, the following conditions apply as well:

- MTS requires that the server be a dynamic-link library (DLL). Servers that are implemented as executable files (.EXE files) cannot execute in the MTS runtime environment.
- The object must implement the *IObjectControl* interface. Support for this interface is automatically added by the Transactional Object wizard.
- A server running in the MTS process space cannot aggregate with COM objects not running in MTS.

Managing resources

Transactional objects can be administered to better manage the resources used by your application. These resources include everything from the memory for the object instances themselves to any resources they use (such as database connections).

In general, you configure how your application manages resources by the way you install and configure your object. You set your transactional object so that it takes advantage of the following:

- Just-in-time activation
- Resource pooling
- Object pooling (COM+ only)

If you want your object to take full advantage of these services, however, it must use the *IObjectContext* interface to indicate when resources can safely be released.

Accessing the object context

As with any COM object, a transactional object must be created before it is used. COM clients create an object by calling the COM library function, *CoCreateInstance*.

Each transactional object must have a corresponding context object. This context object is implemented automatically by MTS or COM+ and is used to manage the transactional object. The context object's interface is *IObjectContext*. To access most methods of the object context, you can use the *ObjectContext* property of the *TMtsAutoObject* object. For example, you can use the *ObjectContext* property as follows:

```
if ObjectContext.IsCallerInRole ('Manager') ...
```

Another way to access the Object context is to use methods in the *TMtsAutoObject* object:

```
if IsCallerInRole ('Manager') ...
```

You can use either of the above methods. However, there is a slight advantage of using the *TMtsAutoObject* methods rather than referencing the ObjectContext property when you are testing your application. For a discussion of the differences, see "Debugging and testing transactional objects" on page 39-21.

Just-in-time activation

The ability for an object to be deactivated and reactivated while clients hold references to it is called **just-in-time activation**. From the client's perspective, only a single instance of the object exists from the time the client creates it to the time it is finally released. Actually, it is possible that the object has been deactivated and reactivated many times. By having objects deactivated, clients can hold references to the object for an extended time without affecting system resources. When an object is deactivated, all its resources can be released. For example, when an object is deactivated, it can release its database connection so that other objects can use it.

A transactional object is created in a deactivated state and becomes active upon receiving a client request. When the transactional object is created, a corresponding context object is also created. This context object exists for the entire lifetime of the transactional object, across one or more reactivation cycles. The context object, accessed by the *IObjectContext* interface, keeps track of the object during deactivation and coordinates transactions.

Transactional objects are deactivated as soon as it is safe to do so. This is called **as-soon-as-possible deactivation**. A transactional object is deactivated when any of the following occurs:

- **The object requests deactivation with *SetComplete* or *SetAbort*:** An object calls the *IObjectContext SetComplete* method when it has successfully completed its work and it does not need to save the internal object state for the next call from the client. An object calls *SetAbort* to indicate that it cannot successfully complete its work and its object state does not need to be saved. That is, the object's state rolls back to the state prior to the current transaction. Often, objects can be designed to be **stateless**, which means that objects deactivate upon return from every method.

- **A transaction is committed or aborted:** When an object's transaction is committed or aborted, the object is deactivated. Of these deactivated objects, the only ones that continue to exist are the ones that have references from clients outside the transaction. Subsequent calls to these objects reactivate them and cause them to execute in a new transaction.
- **The last client releases the object:** Of course, when a client releases the object, the object is deactivated, and the object context is also released.

Note If you install the transactional object under COM+ from the IDE, you can specify whether object supports just-in-time activation using the COM+ page of the Type Library editor. Just select the object (CoClass) in the Type Library editor, go to the COM+ page, and check or uncheck the box for Just In Time Activation. Otherwise, a system administrator specifies this attribute using the COM+ Component Manager or MTS Explorer. (The system administrator can also override any settings you specify using the Type Library editor.)

Resource pooling

Since idle system resources are freed during a deactivation, the freed resources are available to other server objects. For example, a database connection that is no longer used by a server object can be reused by another client. This is called **resource pooling**. Pooled resources are managed by a resource dispenser.

A resource dispenser caches resources, so that transactional objects that are installed together can share them. The resource dispenser also manages nondurable shared state information. In this way, resource dispensers are similar to resource managers such as the SQL Server, but without the guarantee of durability.

When writing your transactional object, you can take advantage of two types of resource dispenser that are provided for you already:

- Database resource dispensers
- Shared Property Manager

Before other objects can use pooled resources, you must explicitly release them.

Database resource dispensers

Opening and closing connections to a database can be time-consuming. By using a resource dispenser to pool database connections, your object can reuse existing database connections rather than create new ones. For example, if you have a database lookup and a database update component running in a customer maintenance application, you can install those components together, and then they can share database connections. In this way, your application does not need as many connections and new object instances can access the data more quickly by using a connection that is already open but not in use.

- If you are using BDE components to connect to your data, the resource dispenser is the Borland Database Engine (BDE). This resource dispenser is only available when your transactional object is installed with MTS. To enable the resource dispenser, use the BDE administrator to turn on MTS POOLING in the System/ Init area of the configuration.

- If you are using the ADO database components to connect to your data, the resource dispenser is provided by ADO.

Note There is no built-in resource pooling if you are using InterbaseExpress components for your database access.

For remote transactional data modules, connections are automatically enlisted on an object's transactions, and the resource dispenser can automatically reclaim and reuse connections.

Shared property manager

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. By using the Shared Property Manager, you avoid having to add a lot of code to your application for managing shared data: the Shared Property Manager handles it for you by implementing locks and semaphores to protect shared properties from simultaneous access. The Shared Property Manager eliminates name collisions by providing **shared property groups**, which establish unique name spaces for the shared properties they contain.

To use the Shared Property Manager resource, you first use the *CreateSharedPropertyGroup* helper function to create a shared property group. Then you can write all the properties to that group and read all the properties from that group. By using a shared property group, the state information is saved across all deactivations of a transactional object. In addition, state information can be shared among all transactional objects installed in the same MTS package or COM+ application. You can install transactional objects into a package as described in "Installing transactional objects" on page 39-22.

For objects to share state, they all must run in the same process. If you want instances of different components to share properties, you must install them in the same MTS package or COM+ application. Because there is a risk that administrators may move components from one package to another, it's safest to limit the use of a shared property group to instances of objects that are defined in the same DLL or EXE.

Objects sharing properties must have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same MTS package or COM+ application.

The following example shows how to add code to support the Shared Property Manager in a transactional object:

Example: Sharing properties among transactional object instances

This example creates a property group called MyGroup to contain the properties to be shared among objects and object instances. In this example, there is a Counter property that is shared. It uses the *CreateSharedPropertyGroup* helper function to create the property group manager and property group, and then uses the *CreateProperty* method of the Group object to create a property called Counter.

To get the value of a property, you use the *PropertyByName* method of the Group object as shown below. You can also use the *PropertyByPosition* method.

```
unit Unit1;
interface
uses
  MtsObj, Mtx, ComObj, Project2_TLB;
type
  Tfoobar = class(TMtsAutoObject, Ifoobar)
  private
    Group: ISharedPropertyGroup;
  protected
    procedure OnActivate; override;
    procedure OnDeactivate; override;
    procedure IncCounter;
  end;
implementation
uses ComServ;
{ Tfoobar }
procedure Tfoobar.OnActivate;
var
  Exists: WordBool;
  Counter: ISharedProperty;
begin
  Group := CreateSharedPropertyGroup('MyGroup');
  Counter := Group.CreateProperty('Counter', Exists);
end;
procedure Tfoobar.IncCounter;
var
  Counter: ISharedProperty;
begin
  Counter := Group.PropertyByName['Counter'];
  Counter.Value := Counter.Value + 1;
end;
procedure Tfoobar.OnDeactivate;
begin
  Group := nil;
end;
initialization
  TAutoObjectFactory.Create(ComServer, Tfoobar, Class_foobar, ciMultiInstance, tmApartment);
end.
```

Releasing resources

You are responsible for releasing resources of an object. Typically, you do this by calling the *IObjectContext* methods *SetComplete* and *SetAbort* after servicing a client request. These methods release the resources allocated by the resource dispenser.

At this same time, you must release references to all other resources, including references to other objects (including transactional objects and context objects) and memory held by any instances of the component (freeing the component).

The only time you would not include these calls is if you want to maintain state between client calls. For details, see “Stateful and stateless objects” on page 39-11.

Object pooling

Just as you can pool resources, under COM+ you can also pool objects. When an object is deactivated, COM+ calls the *IObjectControl* interface method, *CanBePooled*, which indicates that the object can be pooled for reuse. If *CanBePooled* is returns *True*, then instead of being destroyed on deactivation, the object is moved to the object pool. It remains in the object pool for a specified timeout period, during which time it is available for use to any client requesting it. Only when the object pool is empty is a new instance of the object created. Objects that return *False* or that do not support the *IObjectControl* interface are destroyed when they are deactivated.

Object pooling is not available under MTS. MTS calls *CanBePooled* as described, but no pooling takes place. If your object will only run under COM+ and you want to allow object pooling, set the object’s *Pooled* property to *True*.

Even if an object’s *CanBePooled* method returns *True*, it can be configured so that COM+ does not move it to the object pool. If you install the transactional object under COM+ from the IDE, you can specify whether COM+ tries to pool the object using the COM+ page of the Type Library editor. Just select the object (CoClass) in the type library editor, go to the COM+ page, and check or uncheck the box for Object Pooling. Otherwise, a system administrator specifies this attribute using the COM+ Component Manager or MTS Explorer.

Similarly, you can configure the time a deactivated object remains in the object pool before it is freed If you are installing from the IDE, you can specify this duration using the Creation Timeout setting on the COM+ page of the type library editor. Otherwise, a system administrator specifies this attribute using the COM+ Component Manager.

MTS and COM+ transaction support

The transaction support that gives transactional objects their name lets you group actions into transactions. For example, in a medical records application, if you had a Transfer component to transfer records from one physician to another, you could include your Add and Delete methods in the same transaction. That way, either the entire Transfer works or it can be rolled back to its previous state. Transactions simplify error recovery for applications that must access *multiple* databases.

Transactions ensure that

- All updates in a single transaction are either committed or get aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial and uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**. Resource managers use transaction-based synchronization protocols to isolate the uncommitted work of active transactions.
- Committed updates to managed resources (such as database records) survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**. Transactional logging allows you to recover the durable state after disk media failures.

An object's associated context object indicates whether the object is executing within a transaction and, if so, the identity of the transaction. When an object is part of a transaction, the services that resource managers and resource dispensers perform on its behalf execute under the transaction as well. Resource dispensers use the context object to provide transaction-based services. For example, when an object executing within a transaction allocates a database connection by using the ADO or BDE resource dispenser, the connection is automatically enlisted on the transaction. All database updates using this connection become part of the transaction, and are either committed or aborted.

Work from multiple objects can be composed into a single transaction. Allowing an object to either live in its own transaction or be part of a larger group of objects that belong to a single transaction is a major advantage of MTS and COM+. It allows an object to be used in various ways, so that application developers can reuse application code in different applications without rewriting the application logic. In fact, developers can determine how objects are used in transactions when installing the transactional object. They can change the transaction behavior simply by adding an object to a different MTS package or COM+ application. For details about installing transactional objects, see “Installing transactional objects” on page 39-22.

Transaction attributes

Every transactional object has a transaction attribute that is recorded in the MTS catalog or that is registered with COM+.

Delphi lets you set the transaction attribute at design time using the Transactional Object wizard or the Type Library editor.

Each transaction attribute can be set to these settings:

Requires a transaction Objects must execute *within the scope of a transaction*. When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction context, a new one is automatically created.

Requires a new transaction Objects must execute *within their own transactions*. When a new object is created, a new transaction is automatically created for the object, regardless of whether its client has a transaction. An object never runs inside the scope of its client's transaction. Instead, the system always creates independent transactions for the new objects.

Supports transactions Objects can execute *within the scope of their client's transactions*. When a new object is created, its object context inherits the transaction from the context of the client. This enables multiple objects to be composed in a single transaction. If the client does not have a transaction, the new context is also created without one.

Transactions Ignored Objects *do not run within the scope of transactions*. When a new object is created, its object context is created without a transaction, regardless of whether the client has a transaction. This setting is only available under COM+.

Does not support transactions The meaning of this setting varies, depending on whether you install the object under MTS or COM+. Under MTS, this setting has the same meaning as Transactions Ignored under COM+. Under COM+, not only is the object context created without a transaction, this setting prevents the object from being activated if the client has a transaction.

Setting the transaction attribute

You can set a transaction attribute when you first create a transactional object using the Transactional Object wizard.

You can also set (or change) the transaction attribute using the Type Library editor. To change the transaction attribute in the Type Library editor,

1 Choose View | Type Library to open the Type Library editor.

2 Select the class corresponding to the transactional object.

3 Click the COM+ tab and choose the desired transaction attribute.

Warning When you set the transaction attribute, Delphi inserts a special GUID for the specified attribute as custom data in the type library. This value is not recognized outside of Delphi. Therefore, it only has an effect if you install the transactional object from the IDE. Otherwise, a system administrator must set this value using the MTS Explorer or COM+ Component Manager.

Note: If the transactional object is already installed, you must first uninstall the object and reinstall it when changing the transaction attribute. Use Run | Install MTS objects or Run | Install COM+ objects to do so.

Stateful and stateless objects

Like any COM object, transactional objects can maintain internal state across multiple interactions with a client. For example, the client could set a property value in one call, and expect that property value to remain unchanged when it makes the next call. Such an object is said to be **stateful**. Transactional objects can also be **stateless**, which means the object does not hold any intermediate state while waiting for the next call from a client.

When a transaction is committed or aborted, all objects that are involved in the transaction are deactivated, causing them to lose any state they acquired during the course of the transaction. This helps ensure transaction isolation and database consistency; it also frees server resources for use in other transactions. Completing a transaction enables the resources held by an object to be reclaimed when the object is deactivated. See the following section for information on how to control when the object's state is released.

Maintaining state on an object requires the object to remain activated, holding potentially valuable resources such as database connections.

Influencing how transactions end

A transactional object uses the *IObjectContext* methods as shown in the following table to influence how a transaction completes. These methods, together with the object's transaction attribute, allow you to enlist one or more objects into a single transaction.

Table 39.1 IObjectContext methods for transaction support

Method	Description
SetComplete	Indicates that the object has successfully completed its work for the transaction. The object is deactivated upon return from the method that first entered the context. The object reactivates on the next call that requires object execution.
SetAbort	Indicates that the object's work can never be committed and the transaction should be rolled back. The object is deactivated upon return from the method that first entered the context. The object reactivates on the next call that requires object execution.

Table 39.1 IObjectContext methods for transaction support (continued)

Method	Description
EnableCommit	Indicates that the object's work is not necessarily done, but that its transactional updates can be committed in their current form. Use this to retain state across multiple calls from a client while still allowing transactions to complete. The object is not deactivated until it calls SetComplete or SetAbort. EnableCommit is the default state when an object is activated. This is why an object should *always call SetComplete or SetAbort before returning from a method,* unless you want the object to maintain its internal state for the next call from a client.
DisableCommit	Indicates that the object's work is inconsistent and that it cannot complete its work until it receives further method invocations from the client. Call this before returning control to the client to maintain state across multiple client calls while keeping the current transaction active. DisableCommit prevents the object from deactivating and releasing its resources on return from a method call. Once an object has called DisableCommit, if a client attempts to commit the transaction before the object has called EnableCommit or SetComplete, the transaction will abort.

Initiating transactions

Transactions can be controlled in three ways:

- They can be controlled by the client.

  Clients can have direct control over transactions by using a transaction context object (using the *ITransactionContext* interface).

- They can be controlled by the server.

  Servers can control transactions explicitly creating an object context for them. When the server creates an object this way, the created object is automatically enlisted in the current transaction.

- Transactions can occur automatically as a result of the object’s transaction attribute.

  Transactional objects can be declared so that their objects always execute within a transaction, regardless of how the objects are created. This way, objects do not need to include any logic to handle transactions. This feature also reduces the burden on client applications. Clients do not need to initiate a transaction simply because the component that they are using requires it.

Setting up a transaction object on the client side

A client-based application can control transaction context through the *ITransactionContextEx* interface. The following code example shows how a client application uses *CreateTransactionContextEx* to create the transaction context. This method returns an interface to this object.

This example wraps the call to the transaction context in a call to OleCheck which is necessary because the methods of *IObjectContext* are exposed by Windows directly and are therefore not declared as **safecall**.

```
procedure TForm1.MoveMoneyClick(Sender: TObject);
begin
  Transfer(CLASS_AccountA, CLASS_AccountB, 100);
end;
procedure TForm1.Transfer(DebitAccountId, CreditAccountId: TGuid; Amount: Currency);
var
  TransactionContextEx: ITransactionContextEx;
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  TransactionContextEx := CreateTransactionContextEx;
  try
    OleCheck(TransactionContextEx.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(TransactionContextEx.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    TransactionContextEx.Abort;
    raise;
  end;
  TransactionContextEx.Commit;
end;
```

Setting up a transaction object on the server side

To control transaction context from the server side, you create an instance of *ObjectContext*. In the following example, the Transfer method is in the transactional object. In using ObjectContext this way, the instance of the object we are creating will inherit all the transaction attributes of the object that creates it. We wrap the call in a call to OleCheck because the methods of *IObjectContext* are exposed by Windows directly and are therefore not declared as **safecall**.

```
procedure TAccountTransfer.Transfer(DebitAccountId, CreditAccountId: TGuid;
 Amount: Currency);
var
  CreditAccountIntf, DebitAccountIntf: IAccount;
begin
  try
    OleCheck(ObjectContext.CreateInstance(DebitAccountId,
      IAccount, DebitAccountIntf));
    OleCheck(ObjectContext.CreateInstance(CreditAccountId,
      IAccount, CreditAccountIntf));
    DebitAccountIntf.Debit(Amount);
    CreditAccountIntf.Credit(Amount);
  except
    DisableCommit;
    raise;
  end;
  EnableCommit;
end;
```

Transaction timeout

The transaction timeout sets how long (in seconds) a transaction can remain active. The system automatically aborts transactions that are still alive after the timeout. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging transactional objects.

To set the timeout value on your computer,

1. In the MTS Explorer or COM+ Component Manager, select Computer, My Computer.

   By default, My Computer corresponds to the local computer.
2. Right-click and choose Properties and then choose the Options tab.

   The Options tab is used to set the computer's transaction timeout property.
3. Change the timeout value to 0 to disable transaction timeouts.
4. Click OK to save the setting.

For more information on debugging MTS applications, see “Debugging and testing transactional objects” on page 39-21.

Role-based security

MTS and COM+ provide role-based security where you assign a role to a logical group of users. For example, a medical information application might define roles for Physician, X-ray technician, and Patient.

You define authorization for each object and interface by assigning roles. For example, in the physicians’ medical application, only the Physician may be authorized to view all medical records; the X-ray Technician may view only X-rays; and Patients may view only their own medical record.

Typically, you define roles during application development and assign roles for each MTS package or COM+ Application. These roles are then assigned to specific users when the application is deployed. Administrators can configure the roles using the MTS Explorer or COM+ Component Manager.

If you want to control access to blocks of code rather than entire objects, you can provide more fine-grained security by using the *IObjectContext* method, *IsCallerInRole*. This method only works if security is enabled, which can be checked by calling the *IObjectContext* method *IsSecurityEnabled*. These methods are automatically added as methods to your transactional object. For example,

```
if IsSecurityEnabled then {check if security is enabled }
begin
  if IsCallerInRole(‘Physician’) then { check caller’s role }
  begin
    { execute the call normally }
  end
  else
```

```
      { not a physician, do something appropriate }
    end
  end
  else
    { no security enabled, do something appropriate }
  end;
```

Note For applications that require stronger security, context objects implement the *ISecurityProperty* interface, whose methods allow retrieval of the Window's security identifier (SID) for the direct caller and creator of the object, as well as the SID for the clients which are using the object.

Overview of creating transactional objects

The process of creating transactional object is as follows:

1. Use the Transactional Object wizard to create the transactional object.
2. Add methods and properties to the object's interface using the Type Library editor. For details on adding methods and properties using the Type Library editor, see Chapter 34, "Working with type libraries."
3. When implementing your object's methods, you can use the *IObjectContext* interface to manage transactions, persistent state, and security. In addition, if you are passing object references, you will need to use extra care so that they are correctly handled. (See "Passing object references" on page 20.)
4. Debug and test the transactional object.
5. Install the transactional object into an MTS package or COM+ application.
6. Administer your objects using the MTS Explorer or COM+ Component Manager.

Using the Transactional Object wizard

Use the Transactional Object wizard to create a COM object that can take advantage of the resource management, transaction processing, and role-based security provided by MTS or COM+.

To bring up the Transactional Object wizard,

1. Choose File | New | Other.
2. Select the tab labeled Multitier.
3. Double-click the Transactional Object icon.

In the wizard, you must specify the following:

- A threading model that indicates how client applications can call your object's interface. The threading model determines how the object is registered. You are responsible for ensuring that the object's implementation adheres to the selected model. For more information on threading models, see "Choosing a threading model for a transactional object" on page 39-16.

- A transaction model
- An indication of whether your object notifies clients of events. Event support is only provided for traditional events, not COM+ events.

When you complete this procedure, a new unit is added to the current project that contains the definition for the transactional object. In addition, the wizard adds a type library to the project and opens it in the Type Library editor. Now you can expose the properties and methods of the interface through the type library. You define the interface as you would define any COM object as described in “Defining a COM object’s interface” on page 36-9.

The transactional object implements a **dual interface**, which supports both early (compile-time) binding through the vtable and late (runtime) binding through the *IDispatch* interface.

The generated transactional object implements the *IObjectControl* interface methods, *Activate*, *Deactivate*, and *CanBePooled*.

It is not strictly necessary to use the transactional object wizard. You can convert any Automation object into a COM+ transactional object (and any in-process Automation object into an MTS transactional object) by using the COM+ page of the Type Library editor and then installing the object into an MTS package or COM+ application. However, the transactional object wizard provides certain benefits:

- It automatically implements the *IObjectControl* interface, adding *OnActivate* and *OnDeactivate* events to the object so that you can create event handlers that respond when the object is activated or deactivated.
- It automatically generates an ObjectContext property so that it is easy for your object to access the *IObjectContext* methods to control activation and transactions.

Choosing a threading model for a transactional object

The MTS runtime environment or COM+ manages threads for you. Transactional objects should not create threads. They must also never terminate a thread that calls into a DLL.

When you specify the threading model using the Transactional object wizard, you specify how objects are assigned to threads for method execution.

Table 39.2 Threading models for transactional objects

Threading model	Description	Implementation pros and cons
Single	No thread support. Client requests are serialized by the calling mechanism. All objects of a single-threaded component execute on the main thread. This is compatible with the default COM threading model, which is used for components that do not have a Threading Model Registry attribute or for COM components that are not reentrant. Method execution is serialized across all objects in the component and across all components in a process.	Allows components to use libraries that are not reentrant. Very limited scalability. Single-threaded, stateful components are prone to deadlocks. You can eliminate this problem by using stateless objects and calling SetComplete before returning from any method.
Apartment (or Single-threaded apartment)	Each object is assigned to a thread apartment, which lasts for the life of the object; however, multiple threads can be used for multiple objects. This is a standard COM concurrency model. Each apartment is tied to a specific thread and has a Windows message pump.	Provides significant concurrency improvements over the single threading model. Two objects can execute concurrently as long as they are not in the same activity. Similar to a COM apartment, except that the objects can be distributed across multiple processes.

Note These threading models are similar to those defined by COM objects. However, because the MTS and COM+ provide more underlying support for threads, the meaning of each threading model differs here. Also, the free threading model does not apply to transactional objects due to the built-in support for activities.

Activities

In addition to the threading model, transactional objects achieve concurrency through **activities**. Activities are recorded in an object's context, and the association between an object and an activity cannot be changed. An activity includes the transactional object created by the base client, as well as any transactional objects created by that object and its descendants. These objects can be distributed across one or more processes, executing on one or more computers.

For example, a physician's medical application may have a transactional object to add updates and remove records to various medical databases, each represented by a different object. This record object may use other objects as well, such as a receipt object to record the transaction. This results in several transactional objects that are either directly or indirectly under the control of a base client. These objects all belong to the same activity.

MTS or COM+ tracks the flow of execution through each activity, preventing inadvertent parallelism from corrupting the application state. This feature results in a single logical thread of execution throughout a potentially distributed collection of objects. By having one logical thread, applications are significantly easier to write.

When a transactional object is created from an existing context, using either a transaction context object or an object context, the new object becomes a member of the same activity. In other words, the new context inherits the activity identifier of the context used to create it.

Only a single logical thread of execution is allowed within an activity. This is similar in behavior to a COM apartment threading model, except that the objects can be distributed across multiple processes. When a base client calls into an activity, all other requests for work in the activity (such as from another client thread) are blocked until after the initial thread of execution returns back to the client.

Under MTS, every transactional object belongs to one activity. Under COM+, you can configure the way the object participates in activities by setting the **call synchronization**. The following options are available:

Table 39.3 Call synchronization options

Option	Meaning
Disabled	COM+ does not assign activities to the object but it may inherit them with the caller's context. If the caller has no transaction or object context, the object is not assigned to an activity. The result is the same as if the object was not installed in a COM+ application. This option should not be used if any object in the application uses a resource manager or if the object supports transactions or just-in-time activation.
Not Supported	COM+ never assigns the object to an activity, regardless of the status of its caller. This option should not be used if any object in the application uses a resource manager or if the object supports transactions or just-in-time activation.
Supported	COM+ assigns the object to the same activity as its caller. If the caller does not belong to an activity, the object does not either. This option should not be used if any object in the application uses a resource manager or if the object supports transactions or just-in-time activation.
Required	COM+ always assigns the object to an activity, creating one if necessary. This option must be used if the transaction attribute is Supported or Required.
Requires New	COM+ always assigns the object to a new activity, which is distinct from its caller's.

Generating events under COM+

Before COM+, Automation servers used a set of special interfaces for generating events. COM+, however, introduces a new system for managing events. Instead of the server object managing events, keeping track of clients that need to be notified and calling their interfaces when events occur, the underlying system (COM+) manages this process.

Note Transactional objects installed under COM+ can still use the old system for managing events. However, letting COM+ handle the process provides greater flexibility. For example, when COM+ manages events, the client can be an in-process server that is launched by COM+ when the event occurs.

When a COM+ object generates events, it does not do so directly. Rather, it makes use of an associated event object that is specifically created to generate events. The COM+ object calls its event object when it wants to fire an event. When that happens, COM+ calls all clients that have registered an interest in the particular event object.

Using the Event Object wizard

You can create event objects using the Event Object wizard. The wizard first checks whether the current project contains any implementation code, because projects containing COM+ event objects do not include an implementation. They can only contain event object definitions. (You can, however, include multiple COM+ event objects in a single project.)

To bring up the Event Object wizard,

1. Choose File | New.
2. Select the tab labeled ActiveX.
3. Double-click the COM+ Event Object icon.

In the Event Object wizard, specify the name of the event object, the name of the interface that defines the event handlers, and (optionally) a brief description of the events.

When you exit, the wizard creates a project containing a type library that defines your event object and its interface. Use the Type Library editor to define the methods of that interface. These methods are the event handlers that clients implement to respond to events.

The Event object project includes the project file, _ATL unit to import the ATL template classes, and the _TLB unit to define the type library information. It does not include an implementation unit, however, because COM+ event objects have no implementation. The implementation of the interface is the responsibility of the client. When your server object calls a COM+ event object, COM+ intercepts the call and dispatches it to registered clients. Because COM+ event objects require no implementation object, all you need to do after defining the object's interface in the Type Library editor is compile the project and install it with COM+

COM+ places certain restrictions on the interfaces of event objects. The interface you define in the Type Library editor for your event object must obey the following rules:

- The event object's interface must derive from IDispatch.
- All method names must be unique across all interfaces of the event object.
- All methods on the event object's interface must return an HRESULT value.
- The modifier for all parameters of methods must be blank.

Firing events using a COM+ event object

When an event occurs, your COM+ object must call the event object and tell it to fire the event on registered clients. It does this by creating an instance of the event object and calling the method that corresponds to the event:

Note Objects that fire COM+ events, like the event objects themselves, must be installed in a COM+ application.

Passing object references

Note Information on passing object references applies only to MTS, not COM+. This mechanism is needed under MTS because it is necessary to ensure that all pointers to objects running under MTS are routed through interceptors. Because interceptors are built into COM+, you do not need to pass object references.

Under MTS, you can pass object references, (for example, for use as a callback) only in the following ways:

- Through return from an object creation interface, such as *CoCreateInstance* (or its equivalent), *ITransactionContext.CreateInstance*, or *IObjectContext.CreateInstance.*
- Through a call to *QueryInterface*.
- Through a method that has called *SafeRef* to obtain the object reference.

An object reference that is obtained in the above ways is called a **safe reference**. Methods invoked using safe references are guaranteed execute within the correct context.

The MTS runtime environment requires calls to use safe references so that it can manage context switches and allows transactional objects to have lifetimes that are independent of client references. Safe references are not necessary under COM+.

Using the SafeRef method

An object can use the *SafeRef* function to obtain a reference to itself that is safe to pass outside its context. The unit that defines the *SafeRef* function is Mtx.

SafeRef takes as input

- A reference to the interface ID (RIID) of the interface that the current object wants to pass to another object or client.
- A reference to the current object's IUnknown interface.

SafeRef returns a pointer to the interface specified in the RIID parameter that is safe to pass outside the current object's context. It returns **nil** if the object is requesting a safe reference on an object other than itself, or the interface requested in the RIID parameter is not implemented.

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a callback), it should always call *SafeRef* first and then pass the reference returned by this call. An object should never pass a **self** pointer, or a self-reference obtained through an internal call to QueryInterface, to a client or to any other object. Once such a reference is passed outside the object's context, it is no longer a valid reference.

Calling *SafeRef* on a reference that is already safe returns the safe reference unchanged, except that the reference count on the interface is incremented.

When a client calls QueryInterface on a reference that is safe, the reference returned to the client is also a safe reference.

An object that obtains a safe reference must release the safe reference when finished with it.

For details on *SafeRef* see the SafeRef topic in the Microsoft documentation.

Callbacks

Objects can make callbacks to clients and to other transactional objects. For example, you can have an object that creates another object. The creating object can pass a reference of itself to the created object; the created object can then use this reference to call the creating object.

If you choose to use callbacks, note the following restrictions:

- Calling back to the base client or another package requires access-level security on the client. Additionally, the client must be a DCOM server.
- Intervening firewalls may block calls back to the client.
- Work done on the callback executes in the environment of the object being called. It may be part of the same transaction, a different transaction, or no transaction.
- Under MTS, the creating object must call *SafeRef* and pass the returned reference to the created object in order to call back to itself.

Debugging and testing transactional objects

You can debug local and remote transactional objects. When debugging transactional objects, you may want to turn off transaction timeouts.

The transaction timeout sets how long (in seconds) a transaction can remain active. Transactions that are still alive after the timeout are automatically aborted by the system. By default, the timeout value is 60 seconds. You can disable transaction timeouts by specifying a value of 0, which is useful when debugging.

For information on remote debugging, see the Remote Debugging topic in Online help.

When testing a transactional object that you intend to run under MTS, you may first want to test your object outside the MTS environment to simplify your test environment.

While developing your server, you cannot rebuild the server when it is still in memory. You may get a compiler error like, “Cannot write to DLL while executable is loaded." To avoid this, you can set the MTS package or COM+ application properties to shut down the server when it is idle.

To shut down the server when idle,

1 In the MTS Explorer or COM+ Component Manager, right-click the MTS package or COM+ application in which your transactional object is installed and choose Properties.

2 Select the Advanced tab.

The Advanced tab determines whether the server process associated with a package always runs, or whether it shuts down after a certain period of time.

3 Change the timeout value to 0, which shuts down the server as soon as no longer has a client to service.

4 Click OK to save the setting.

Note When testing outside the MTS environment, you do not reference the *ObjectProperty* of *TMtsObject* directly. The *TMtsObject* implements methods such as *SetComplete* and *SetAbort* that are safe to call when the object context is **nil**.

Installing transactional objects

MTS applications consist of a group of in-process MTS objects running in a single instance of the MTS executive (EXE). A group of COM objects that all run in the same process is called a **package**. A single machine can be running several different packages, where each package is running within a separate MTS EXE.

Under COM+, you work with a similar group, called a COM+ application. In a **COM+ application**, the objects need not be in-process, and there is no separate runtime environment.

You can group your application components into a single MTS package or COM+ application to be managed by a single process. You might want to distribute your components into different MTS packages or COM+ applications to partition your application across multiple processes or machines.

To install transactional objects into an MTS package or COM+ application,

1 If your system supports COM+, choose Run | Install COM+ objects. If your system does not support COM+ but you have MTS installed on your system, choose Run | Install MTS objects. If your system supports neither MTS nor COM+, you will not see a menu item for installing transactional objects.

2 In the Install Object dialog box, check the objects to be installed.

3 If you are installing MTS objects, click the Package button to get a list of MTS packages on your system. If you are installing COM+ objects, click the Application button. Indicate the MTS package or COM+ application into which you are installing your objects. You can choose Into New Package or Into New Application to create a new MTS package or COM+ application in which to install the object. You can choose Into Existing Package or Into Existing Application to install the object into an existing listed MTS package or COM+ application.

4 Choose OK to refresh the catalog, which makes the objects available at runtime.

MTS packages can contain components from multiple DLLs, and components from a single DLL can be installed into different packages. However, a single component cannot be distributed among multiple packages.

Similarly, COM+ applications can contain components from multiple executables and different components from a single executable can be installed into different COM+ applications.

Note You can also install your transactional object using the COM+ Component Manager or MTS Explorer. Be sure when installing the object with one of these tools that you apply the settings for the object that appear on the COM+ page of the Type Library editor. These settings are not applied automatically when you do not install from the IDE.

Administering transactional objects

Once you have installed transactional objects, you can administer these runtime objects using the MTS Explorer (if they are installed into an MTS package) or the COM+ Component Manager (if they are installed into a COM+ application). Both tools are identical, except that the MTS Explorer operates on the MTS runtime environment and the COM+ Component Manager operates on COM+ objects.

The COM+ Component Manager and MTS Explorer have a graphical user interface for managing and deploying transactional objects. Using one of these tools, you can

- Configure transactional objects, MTS packages or COM+ applications, and roles
- View properties of components in an package or COM+ application and view the MTS packages or COM+ applications installed on a computer
- Monitor and manage transactions for objects that comprise transactions
- Move MTS packages or COM+ applications between computers
- Make a remote transactional object available to a local client

For more details on these tools, see the appropriate *Administrator's Guide* from Microsoft.

Part

V

Creating custom components

The chapters in “Creating custom components” present concepts necessary for designing and implementing custom components in Delphi.

Chapter

40

Overview of component creation

This chapter provides an overview of component design and the process of writing components for Delphi applications. The material here assumes that you are familiar with Delphi and its standard components.

- VCL and CLX
- Components and classes
- How do you create components?
- What goes into a component?
- Creating a new component
- Testing uninstalled components
- Testing installed components

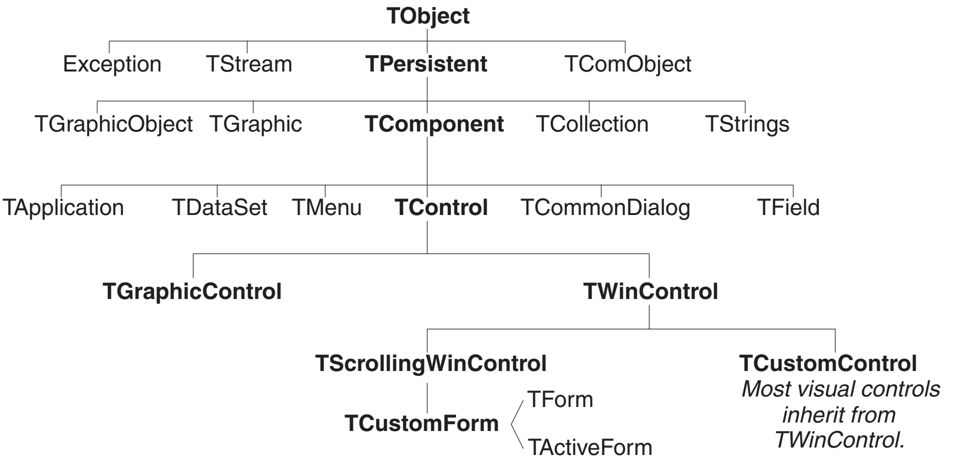
For information on installing new components, see “Installing component packages” on page 11-5.

VCL and CLX

Delphi’s components reside in two class hierarchies called the Visual Component Library (VCL) and the Component Library for Cross Platform (CLX). Figure 40.1 shows the relationship of selected classes that make up the VCL. The CLX hierarchy is similar to the VCL but Windows controls are called widgets (therefore *TWinControl* is called *TWidgetControl*, for example), and there are other differences. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Chapter 41, “Object-oriented programming for component writers.” For an overview of how CLX differs from the VCL, see “CLX versus VCL” on page 10-5 and refer to the CLX online reference for details on the components.

The *TComponent* class is the shared ancestor of every component in the VCL and CLX. *TComponent* provides the minimal properties and events necessary for a component to work in Delphi. The various branches of the library provide other, more specialized capabilities.

Figure 40.1 Visual Component Library class hierarchy



When you create a component, you add to the VCL or CLX by deriving a new class from one of the existing class types in the hierarchy.

Components and classes

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

Briefly, there are two main differences between creating components and using them in applications. When creating components,

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

How do you create components?

A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one. You can derive a new component from any existing component, but the following are the most common ways to create components:

- Modifying existing controls
- Creating windowed controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

Table 40.1 summarizes the different kinds of components and the classes you use as starting points for each.

Table 40.1 Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TListBox*, or an abstract component type, such as *TCustomListBox*
Create a windowed (or widget-based in CLX) control	*TWinControl* (*TWidgetControl* in CLX)
Create a graphic control	*TGraphicControl*
Subclassing a control	Any Windows (VCL) or widget-based (CLX) control
Create a nonvisual component	*TComponent*

You can also derive classes that are not components and cannot be manipulated on a form. Delphi includes many such classes, like *TRegIniFile* and *TFont*.

Modifying existing controls

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided with Delphi.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the VCL and CLX includes an abstract class (with the word “custom” in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWinControl* (or *TWidgetControl* in CLX) class and reinvent all the list box functions, the VCL or CLX provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you publish only the properties you want to make available in your component and leave the rest protected.

Chapter 42, “Creating properties,” explains publishing inherited properties. Chapter 48, “Modifying an existing component,” and Chapter 50, “Customizing a grid,” show examples of modifying existing controls.

Creating windowed controls

Windowed controls in the VCL and CLX are objects that appear at runtime and that the user can interact with. Each windowed control has a window handle, accessed through its *Handle* property, that lets the operating system identify and operate on the control. If using VCL controls, the handle allows the control to receive input focus and can be passed to Windows API functions. In CLX, these controls are widget-based controls. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.

All windowed controls descend from the *TWinControl* (*TWidgetControl* in CLX) class. These include most standard windowed controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that's not related to any existing control) directly from *TWinControl* (*TWidgetControl* in CLX), Delphi provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized windowed control that makes it easier to draw complex visual images.

Chapter 50, "Customizing a grid," presents an example of creating a windowed control.

Creating graphic controls

If your control does not need to receive input focus, you can make it a graphic control. Graphic controls are similar to windowed controls, but have no window handles, and therefore consume fewer system resources. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to mouse messages.

Delphi supports the creation of custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on and on Windows, handles *WM_PAINT* messages; all you need to do is override the *Paint* method.

Chapter 49, "Creating a graphic component," presents an example of creating a graphic control.

Subclassing Windows controls

In traditional Windows programming, you create custom controls by defining a new *window class* and registering it with Windows. The window class (which is similar to the *objects* or *classes* in object-oriented programming) contains information shared among instances of the same sort of control; you can base a new window class on an existing class, which is called *subclassing*. You then put your control in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

Using Delphi, you can create a component "wrapper" around any existing window class. So if you already have a library of custom controls that you want to use in Delphi applications, you can create Delphi components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing Windows controls, see the components in the StdCtls unit that represent standard Windows controls, such as *TEdit*. For CLX examples, see QStdCtls.

Creating nonvisual components

Nonvisual components are used as interfaces for elements like databases (*TDataSet* or *TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TCommonDialog* (VCL) or *TDialog* (CLX) and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components.

What goes into a component?

To make your components reliable parts of the Delphi environment, you need to follow certain conventions in their design. This section discusses the following topics:

- Removing dependencies
- Properties, methods, and events
- Graphics encapsulation
- Registration

Removing dependencies

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An excellent example of removing dependencies is the *Handle* property of *TWinControl*. If you have written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a window or control until you have created it by calling the *CreateWindow* API function. Delphi windowed controls relieve users from this concern by ensuring that a valid window handle is always available when needed. By using a property to represent the window handle, the control can check whether the window has been created; if the handle is not valid, the control creates a window and returns the handle. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating the window, Delphi components allow developers to focus on what they really want to do. Before passing a window handle to an API function, there is no need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

Properties, methods, and events

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a chapter devoted to it in this book, but the discussion that follows explains some of the motivation for their use.

Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

Chapter 42, "Creating properties," explains how to add properties to your components.

Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

Chapter 43, "Creating events," explains how to implement standard events and how to define new ones.

Methods

Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component's constructor method (*Create*) is a class method. Component methods are procedures and functions that operate on the component instances themselves. Application developers use methods to direct a component to perform a specific action or return a value not contained by any property.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

Chapter 44, "Creating methods," explains how to add methods to your components.

Graphics encapsulation

Delphi simplifies Windows graphics by encapsulating various graphic tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is like a Windows device context, but it takes care of all the bookkeeping for you.

If you have written a graphical Windows application, you are familiar with the requirements imposed by Windows' graphics device interface (GDI). For example, GDI limits the number of device contexts available and requires that you restore graphic objects to their initial state before destroying them.

With Delphi, you do not have to worry about these things. To draw on a form or other component, you access the component's *Canvas* property. If you want to customize a pen or brush, you set its color or style. When you finish, Delphi disposes of the resources. Delphi caches resources to avoid recreating them if your application frequently uses the same kinds of resource.

You still have full access to the Windows GDI, but you will often find that your code is simpler and runs faster if you use the canvas built into Delphi components. Graphics features are detailed in Chapter 45, "Using graphics in components."

CLX graphics encapsulation works differently. A canvas is a painter instead. To draw on a form or other component, you access the component's *Canvas* property. *Canvas* is a property and it is also an object called *TCanvas*. *TCanvas* is a wrapper around a Qt painter that is accessible through the *Handle* property. You can use the handle to access low-level Qt graphics library functions.

If you want to customize a pen or brush, you set its color or style. When you finish, Kylix disposes of the resources. CLX also caches the resources.

You can use the canvas built into CLX components by descending from them. How graphics images work in the component depends on the canvas of the object from which your component descends.

Registration

Before you can install your components in the Delphi IDE, you have to register them. Registration tells Delphi where to place the component on the Component palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Chapter 47, “Making components available at design time.”

Creating a new component

You can create a new component two ways:

- Using the Component wizard
- Creating a component manually

You can use either of these methods to create a minimally functional component ready to install on the Component palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the Component palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

1. Create a unit for the new component.
2. Derive your component from an existing component type.
3. Add properties, methods, and events.
4. Register your component with Delphi.
5. Create a Help file for your component and its properties, methods, and events.
6. Create a package (a special dynamic-link library) so that you can install your component in the Delphi IDE.

When you finish, the complete component includes the following files:

- A package (.BPL) or package collection (.DPC) file
- A compiled package (.DCP) file
- A compiled unit (.DCU) file
- A palette bitmap (.DCR) file
- A Help (.HLP) file

Creating a help file to instruct component users on how to use the component is optional.

The chapters in the rest of Part V explain all the aspects of building components and provide several complete examples of writing different kinds of components.

Using the Component wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify only these things:

- The class from which it is derived
- The class name for the new component
- The Component palette page where you want it to appear
- The name of the unit in which the component is created
- The search path where the unit is found
- The name of the package in which you want to place the component

The Component wizard performs the same tasks you would when creating a component manually:

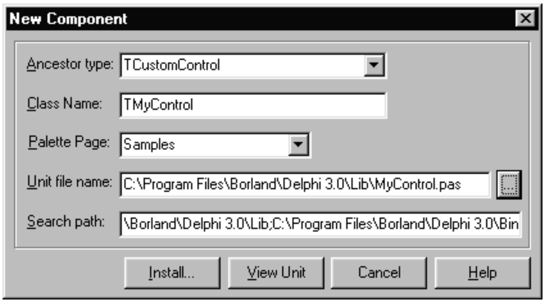
- Creating a unit
- Deriving the component
- Registering the component

The Component wizard cannot add components to an existing unit. You must add components to existing units manually.

To start the Component wizard, choose one of these two methods:

- Choose Component | New Component.
- Choose File | New | Other and double-click on Component

Figure 40.2 Component wizard



Ancestor type: TCustomControl

Class Name: TMyControl

Palette Page: Samples

Unit file name: C:\Program Files\Borland\Delphi 3.0\Lib\MyControl.pas

Search path: \Borland\Delphi 3.0\Lib;C:\Program Files\Borland\Delphi 3.0\Bin

Fill in the fields in the Component wizard:

1 In the Ancestor Type field, specify the class from which you are deriving your new component.

Note In the drop-down list, many components are listed twice with different unit names, one for VCL and one for CLX. The CLX-specific units begin with Q (such as QGraphics instead of Graphics). Be sure to descend from the correct component.

2 In the Class Name field, specify the name of your new component class.

3 In the Palette Page field, specify the page on the Component palette on which you want the new component to be installed.

4 In the Unit file name field, specify the name of the unit you want the component class declared in.

5 If the unit is not on the search path, edit the search path in the Search Path field as necessary.

To place the component in a new or existing package, click Component|Install and use the dialog box that appears to specify a package.

Warning If you derive a component from a VCL or CLX class whose name begins with "custom" (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. Delphi cannot create instance objects of a class that has abstract properties or methods.

To see the source code for your unit, click View Unit. (If the Component wizard is already closed, open the unit file in the Code editor by selecting File|Open.) Delphi creates a new unit containing the class declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Delphi units.

The unit looks like this if descending from *TCustomControl* in the Controls unit:

```
unit MyControl;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TMyControl = class(TCustomControl)
  private
  { Private declarations }
  protected
  { Protected declarations }
  public
  { Public declarations }
  published
  { Published declarations }
end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyControl]);
end;

end.
```

If descending from *TCustomControl* in the QControls unit, the only difference is the **uses** clause which looks like this:

```
uses
  Windows, Messages, SysUtils, Classes, QControls;
```

Where CLX uses separate units, they are replaced with units of the same name prefixed with a Q; Controls is replaced by QControls.

Creating a component manually

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

To create a component manually, follow these steps:

1 Creating a unit file
2 Deriving the component
3 Registering the component

Creating a unit file

A unit is a separately compiled module of Object Pascal code. Delphi uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well

When you create a component, you either create a new unit for the component or add the new component to an existing unit.

To create a unit, choose File | New | Unit. Delphi creates a new unit file and opens it in the Code editor.

To open an existing unit, choose File | Open and select the source code unit that you want to add your component to.

Note When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the Component palette.

Once you have either a new or existing unit for your component, you can derive the component class.

Deriving the component

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. “How do you create components?” on page 40-2 describes which class to derive different kinds of components from.

Deriving classes is explained in more detail in the section “Defining new classes” on page 41-1.

To derive a component, add an object type declaration to the **interface** part of the unit that will contain the component.

A simple component class is a nonvisual component descended directly from *TComponent*.

To create a simple component class, add the following class declaration to the **interface** part of your component unit:

```
type
  TNewComponent = class(TComponent)
  end;
```

So far the new component does nothing different from *TComponent*. You have created a framework on which to build your new component.

Registering the component

Registration is a simple process that tells Delphi which components to add to its component library, and on which pages of the Component palette they should appear. For a more detailed discussion of the registration process, see Chapter 47, "Making components available at design time."

To register a component,

1 Add a procedure named *Register* to the **interface** part of the component's unit. *Register* takes no parameters, so the declaration is very simple:

```
procedure Register;
```

If you are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not need to change the declaration.

2 Write the *Register* procedure in the **implementation** part of the unit, calling *RegisterComponents* for each component you want to register. *RegisterComponents* is a procedure that takes two parameters: the name of a Component palette page and a set of component types. If you are adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

To register a component named *TMyControl* and place it on the Samples page of the palette, you would add the following *Register* procedure to the unit that contains *TMyControl*'s declaration:

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewControl]);
end;
```

This *Register* procedure places *TMyControl* on the Samples page of the Component palette.

Once you register a component, you can compile it into a package (see Chapter 47, "Making components available at design time") and install it on the Component palette.

Testing uninstalled components

You can test the runtime behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette. For information on testing already installed components, see "Testing installed components" on page 40-14.

You test an uninstalled component by emulating the actions performed by Delphi when the component is selected from the palette and placed on a form.

To test an uninstalled component,

1 Add the name of component's unit to the form unit's **uses** clause.

2 Add an object field to the form to represent the component.

This is one of the main differences between the way you add components and the way Delphi does it. You add the object field to the public part at the bottom of the form's type declaration. Delphi would add it above, in the part of the type declaration that it manages.

Never add fields to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.

3 Attach a handler to the form's *OnCreate* event.

4 Construct the component in the form's *OnCreate* handler.

When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You will nearly always pass *Self* as the owner. In a method, *Self* is a reference to the object that contains the method. In this case, in the form's *OnCreate* handler, *Self* refers to the form.

5 Assign the *Parent* property.

Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that contains the control visually; usually it is the form on which the control appears, but it might be a group box or panel. Normally, you'll set *Parent* to *Self*, that is, the form. Always set *Parent* before setting other properties of the control.

Warning If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating-system problem.

6 Set any other component properties as desired.

Suppose you want to test a new component of type *TMyControl* in a unit named *MyControl*. Create a new project, then follow the steps to end up with a form unit that looks like this:

```
unit Unit1;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MyControl;                               { 1. Add NewTest to uses clause }

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);               { 3. Attach a handler to OnCreate
}
  private
    { Private declarations }
  public
```

```
    { Public Declarations }
    MyControl1: TMyControl1;                                      { 2. Add an object field }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyControl1 := TMyControl.Create(Self);                    { 4. Construct the component }
  MyControl1.Parent := Self;          { 5. Set Parent property if component is a control }
  MyControl1.Left := 12;                                    { 6. Set other properties... )
  ⋮                                                                  ...continue as needed }
end;
end.
```

Testing installed components

You can test the design-time behavior of a component after you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette. For information on testing components that have not yet been installed, see “Testing uninstalled components” on page 40-12.

Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.

Test an installed component using a second running instance of Delphi:

1 From the Delphi IDE menu select Project|Options|and on the Directories/ Conditionals page, set the Debug Source Path to the component’s source file.

2 Then select Tools|Debugger Options. On the Language Exceptions, page enable the exceptions you want to track.

3 Open the component source file and set breakpoints.

4 Select Run|Parameters and set the Host Application field to the name and location of the Delphi executable file.

5 In the Run Parameters dialog, click the Load button to start a second instance of Delphi.

6 Then drop the components to be tested on the form, which should break on your breakpoints in the source.

Chapter

41

Object-oriented programming for component writers

If you have written applications with Delphi, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in this chapter, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes
- Ancestors, descendants, and class hierarchies
- Controlling access
- Dispatching methods
- Abstract class members
- Classes and pointers

Defining new classes

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

Deriving new classes

There are two reasons to derive a new class:

- To change class defaults to avoid repetition
- To add new capabilities to a class

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

To change class defaults to avoid repetition

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Component palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Chapter 48, “Modifying an existing component,” shows an example of changing a component’s default properties.

Note If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

To add new capabilities to a class

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component’s ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Chapter 50, "Customizing a grid," shows an example of customizing an abstract component class.

Declaring a new component class

In addition to standard components, Delphi provides many abstract classes designed as bases for deriving new components. Table 40.1 on page 40-3 shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's unit file.

Here is the declaration of a simple graphical component:

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

A finished component declaration usually includes property, event, and method declarations before the **end**. But a declaration like the one above is also valid, and provides a starting point for the addition of component features.

Ancestors, descendants, and class hierarchies

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, Delphi derives your component from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling access

There are five levels of *access control*—also called *visibility*—on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

Table 41.1 shows the levels of visibility, from most restrictive to most accessible:

Table 41.1 Levels of visibility within an object

Visibility	Meaning	Used for
private	Accessible only to code in the unit where the class is defined.	Hiding implementation details.
protected	Accessible to code in the unit(s) where the class and its descendants are defined.	Defining the component writer's interface.
public	Accessible to all code.	Defining the runtime interface.
automated	Accessible to all code. Automation type information is generated.	OLE automation only.
published	Accessible to all code and from the Object Inspector.	Defining the design-time interface.

Declare members as **private** if you want them to be available only within the class where they are defined; declare them as **protected** if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

Hiding implementation details

Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

Here is an example that shows how declaring a field as **private** hides it from application developers. The listing shows two VCL form units. Each form has a handler for its *OnCreate* event which assigns a value to a private field. The compiler allows assignment to the field only in the form where it is declared.

```
unit HideInfo;
interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;

type
  TSecretForm = class(TForm)                                      { declare new form }
    procedure FormCreate(Sender: TObject);
  private                                                      { declare private part }
    FSecretCode: Integer;                                    { declare a private field }
  end;

var
  SecretForm: TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;                                         { this compiles correctly }
end;
end.                                                                   { end of unit }

unit TestHide;                                           { this is the main form file }

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo;                                           { use the unit with TSecretForm }

type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;
var
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;        { compiler stops with "Field identifier expected" }
end;
end.                                                                   { end of unit }
```

Although a program using the *HideInfo* unit can use objects of type *TSecretForm*, it cannot access the *FSecretCode* field in any of those objects.

Defining the component writer’s interface

Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files).

You can use **protected** declarations to define a *component writer’s interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

Note A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the component that receives the event. As a result, they do not have access to the component’s protected methods (unless the component is declared in the same unit as the form).

Defining the runtime interface

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

Here is an example that shows two read-only properties declared as part of a component's runtime interface:

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;                           { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;              { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
⋮
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

Defining the design-time interface

Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Read-only properties cannot be part of the design-time interface because the application developer cannot assign values to them directly. Read-only properties should therefore be public, rather than published.

Here is an example of a published property called *Temperature*. Because it is published, it appears in the Object Inspector at design time.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;                           { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;       { writable! }
  end;
```

Dispatching methods

Dispatch refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

Static methods

All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

An example of static methods

In the following code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods inherited from the first component.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;      { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;               { this is also different }
  end;
```

Virtual methods

Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime.

To make a method virtual, add the directive **virtual** after the method declaration. The **virtual** directive creates an entry in the object's *virtual method table*, or VMT, which holds the addresses of all the virtual methods in an object type.

When you derive a new class from an existing one, the new class gets its own VMT, which includes all the entries from the ancestor's VMT plus any additional virtual methods declared in the new class.

Overriding methods

Overriding a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods.

To override a method in a descendant class, add the directive **override** to the end of the method declaration.

Overriding a method causes a compilation error if

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.
- The declarations are not otherwise identical (number and type of arguments parameters differ).

The following code shows the declaration of two simple components. The first declares three methods, each with a different kind of dispatching. The other, derived from the first, replaces the static method and overrides the virtual methods.

```
type
  TFirstComponent = class(TCustomControl)
    procedure Move;               { static method }
    procedure Flash; virtual;     { virtual method }
    procedure Beep; dynamic;      { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;               { declares new method }
    procedure Flash; override;    { overrides inherited method }
    procedure Beep; override;     { overrides inherited method }
  end;
```

Dynamic methods

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the directive **dynamic** after the method declaration.

Abstract class members

When a method is declared as **abstract** in an ancestor class, you must surface it (by redeclaring and implementing it) in any descendant component before you can use the new component in applications. Delphi cannot create instances of a class that contains abstract members. For more information about surfacing inherited parts of classes, see Chapter 42, "Creating properties," and Chapter 44, "Creating methods."

Classes and pointers

Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.

Chapter

42

Creating properties

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining properties
- Creating array properties
- Storing and loading properties

Why create properties?

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.)

Properties provide more power than simple fields because

- Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, you let Delphi read the values from the Object Inspector. The Object Inspector also validates property assignments as soon as they are made.

- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a field can be a call to a method which implements elaborate processing.
- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to *True* sets *Down* property of all other speed buttons in its group to *False*.

Types of properties

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

Table 42.1 How properties appear in the Object Inspector

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value (**true** if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these properties and edit them individually. Object properties must descend from *TPersistent*.
Interface	Properties that are interfaces can appear in the Object Inspector as long as the value is an interface that is implemented by a component (a descendant of *TComponent*). Interface properties often have their own property editors.
Array	Array properties must have their own property editors; the Object Inspector has no built-in support for editing them. You can specify a property editor when you register your components.

Publishing inherited properties

All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published.

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.

If you derive a VCL component from *TWinControl*, for example, it inherits the protected *DockSite* property. By redeclaring *DockSite* in your new component, you can change the level of protection to either public or published.

The following code shows a redeclaration of *DockSite* as published, making it available at design time.

```
type
  TSampleComponent = class(TWinControl)
  published
    property DockSite;
  end;
```

When you redeclare a property, you specify only the property name, not the type and other information described below in “Defining properties”. You can also declare new default values and specify whether to store the property.

Redeclarations can make a property less restricted, but not more restricted. Thus you can make a protected property public, but you cannot hide a public property by redeclaring it as protected.

Defining properties

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

The property declaration

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.
- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a **published** section of the component’s class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read or set in program code.

Here is a typical declaration for a property called *Count*.

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;                        { used for internal storage }
    procedure SetCount (Value: Integer);    { write method }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

Internal data storage

There are no restrictions on how you store the data for a property. In general, however, Delphi components follow these conventions:

- Property data is stored in class fields.
- The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these fields consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

Direct access

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage field without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be updated when the property value changes.

The following component-type declaration shows a property that uses direct access for both the **read** and the **write** parts.

```
type
  TSampleComponent = class(TComponent)
  private                                                { internal storage is private}
    FMyProperty: Boolean;                          { declare field to hold property value }
  published                                        { make property available at design time }
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;
```

Access methods

You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same read and write access methods:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
```

Because each element of the date (day, month, and year) is an int, and because setting each requires encoding the date when set, the code avoids duplication by sharing the read and write methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the read method that obtains the date element:

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);           { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

This is the write method that sets the appropriate date element:

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);        { get current date elements }
    case Index of                                  { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);      { encode the modified date }
    Refresh;                                       { update the visible calendar }
  end;
end;
```

The read method

The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property. For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see "Creating array properties" on page 42-8), both of which pass their index values as parameters. (Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the *Object Pascal Language Guide*.)

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

The write method

The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. (Use index specifiers to create a single write method that is shared by several properties. For more information about index specifiers, see the *Object Pascal Language Guide*.)

If you do not declare a write method, the property is read-only.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is a simple write method for an integer property called *Count* that stores its current value in a field called *FCount*.

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
```

```
      FCount := Value;
      Update;
    end;
  end;
```

Default property values

When you declare a property, you can specify a *default value* for it. Delphi uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, Delphi always stores the property.

To specify a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value. For example,

```
property Cool Boolean read GetCool write SetCool default True;
```

Note Declaring a default value does not set the property to that value. The component's constructor method should initialize property values when appropriate. However, since objects always initialize their fields to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to *False*.

Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append the **nodefault** directive to the property's declaration. For example,

```
property FavoriteFlavor string nodefault;
```

When you declare a property for the first time, there is no need to include **nodefault**. The absence of a declared default value means that there is no default.

Here is the declaration of a component that includes a single Boolean property called *IsTrue* with a default value of *True*. Below the declaration (in the **implementation** section of the unit) is the constructor that initializes the property.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
⋮
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);              { call the inherited constructor }
  FIsTrue := True;                       { set the default value }
end;
```

Creating array properties

Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be fields.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can reference only individual elements of an array property, not the entire range of the property.

The following example shows the declaration of a property that returns a string based on an integer index.

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
⋮
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

Creating properties for subcomponents

By default, when a property’s value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property. However, it is also possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent.

Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents are saved with the component that creates them. In order for this to work, however, the following conditions must be met:

- The *Owner* of the subcomponent must be the component that creates it and uses it as the value of a published property. For subcomponents that are descendants of *TComponent*, you can accomplish this by setting the *Owner* property of the subcomponent. For other subcomponents, you must override the *GetOwner* method of the persistent object so that it returns the creating component.
- If the subcomponent is a descendant of *TComponent*, it must indicate that it is a subcomponent by calling the *SetSubComponent* method. Typically, this call is made either by the owner when it creates the subcomponent or by the constructor of the subcomponent.

Typically, properties whose values are subcomponents are read-only. If you allow a property whose value is a subcomponent to be changed, the property setter must free the subcomponent when another component is assigned as the property value. In addition, the component often re-instantiates its subcomponent when the property is set to nil. Otherwise, once the property is changed to another component, the subcomponent can never be restored at design time. The following example illustrates such a property setter for a property whose value is a *TTimer*:

```
procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
  if Value <> FTimer then
  begin
    if Value <> nil then
    begin
      if (FTimer <> nil and FTimer.Owner = self then
        FTimer.Free;
      FTimer := Value;
      FTimer,FreeNotification(self);
    end
    else { nil value }
    begin
      if FTimer.Owner <> self then
      {
        FTimer := TTimer.Create(self);
        FTimer.SetSubComponent(True);
        FTimer.FreeNotification(self);
      }
    end;
  end;
end;
```

Note that the property setter above called the *FreeNotification* method of the component that is set as the property value. This call ensures that the component that is the value of the property sends a notification if it is about to be destroyed. It sends this notification by calling the *Notification* method. You handle this call by overriding the *Notification* method, as follows:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = FTimer) then
    FTimer := nil;
end;
```

Creating properties for interfaces

You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In the previous topic, the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don't have access to the component that implements that interface. As a result, you can't call its *FreeNotification* method.

To handle this situation, you can call your component's *ReferenceInterface* method:

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

Calling *ReferenceInterface* with a specified interface does the same thing as calling another component's *FreeNotification* method. Thus, after calling *ReferenceInterface* from the property setter, you can override the *Notification* method to handle the notifications from the implementor of the interface:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyInftProp)) then
    MyIntfProp := nil;
end;
```

Note that the *Notification* code assigns **nil** to the *MyIntfProp* property, not to the private field (*FIntfField*). This ensures that *Notification* calls the property setter, which calls *ReferenceInterface* to remove the notification request that was established when the property value was set previously. All assignments to the interface property must be made through the property setter.

Storing and loading properties

Delphi stores forms and their components in form (.dfm in VCL and .xfm in CLX) files. A form file stores the properties of a form and its components. When Delphi developers add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file.

Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties

Using the store-and-load mechanism

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its public and published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

Specifying default values

Delphi components save their property values only if those values differ from the defaults. If you do not specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value.

To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

Note Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value—that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to *False*, pointers to **nil**, and so on. If there is any doubt, assign a value in the constructor method.

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;       { override to set new default }
  published
    property Align default alBottom;                     { redeclare with new default value }
  end;
⋮
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                              { perform inherited initialization }
  Align := alBottom;                                   { assign new default value for Align }
end;
```

Determining what to store

You can control whether Delphi stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property.

To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the name of a Boolean function.

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean function:

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
  ⋮
  published
    property Important: Integer stored True;          { always stored }
    property Unimportant: Integer stored False;       { never stored }
    property Sometimes: Integer stored StoreIt;       { storage depends on function value }
  end;
```

Initializing after loading

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

Note The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component.

The following code comes from the *TDatabase* component. After loading, the database tries to reestablish any connections that were open at the time it was stored, and specifies how to handle any exceptions that occur while connecting.

```
procedure TDatabase.Loaded;
begin
  inherited Loaded;                                    { call the inherited method first}
  try
    if FStreamedConnected then Open                          { reestablish connections }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then                             { at design time... }
      Application.HandleException(Self)                { let Delphi handle the exception }
    else raise;                                                       { otherwise, reraise }
  end;
end;
```

Storing and loading unpublished properties

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Delphi does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Delphi's automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that tells Delphi how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

1 Create methods to store and load the property value.

2 Override the *DefineProperties* method, passing those methods to a filer object.

Creating methods to store and load property values

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Delphi's built-in capabilities for saving and loading simple types. If your property value is built out of types that Delphi knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Delphi knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically created component that is the value of a property named *MyCompProperty*:

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

Overriding the DefineProperties method

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Delphi calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else if MyCompProperty = nil or
         TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
        Result := True
      else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
```

Chapter

43

Creating events

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

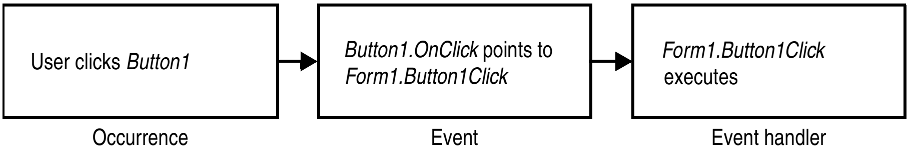
Events are implemented as properties, so you should already be familiar with the material in Chapter 42, “Creating properties,” before you attempt to create or change a component’s events.

What are events?

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance.

From the application developer’s perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* method. By default, Delphi generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.

To write an event, you need to understand the following:


User clicks *Button1*
Button1.OnClick points to *Form1.Button1Click*
Form1.Button1Click executes
Occurrence
Event
Event handler

- Events are method pointers.
- Events are properties.
- Event types are method-pointer types
- Event-handler types are procedures
- Event handlers are optional.

Events are method pointers

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in an instance object. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to an object. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method in a specific instance object. That object is usually the form that contains the component, but it need not be.

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
```

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

Events are properties

Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property.

By convention, the field's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;                 { declare a field to hold the method pointer }
    ⋮
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

To learn about *TNotifyEvent* and other event types, see the next section, "Event types are method-pointer types".

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

Event types are method-pointer types

Because an event is a pointer to an event handler, the type of the event property must be a method-pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of an object.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that is appropriate, or define one of your own.

Event-handler types are procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the *OnKeyPress* event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component may want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

Event handlers are optional

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in “Calling the event” on page 43-8.)

Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component sends numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this should not cause a problem. So the components you create should not require handlers for their events.

Moreover, application developers can write any code they want in an event handler. The components in the VCL and CLX have events written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

Implementing the standard events

The controls that come with Delphi inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events
- Making events visible
- Changing the standard event handling

Identifying standard events

There are two categories of standard events: those defined for all controls and those defined only for the standard windowed controls.

Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether windowed, graphical, or custom, inherit these events. The following events are available in all controls:

OnClick
OnDblClick
OnDragDrop
OnDragOver
OnEndDrag
OnMouseDown
OnMouseMove
OnMouseUp

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named *DoEndDrag*.

Standard events for standard controls

In addition to the events common to all controls, standard windowed controls (those that descend from *TWinControl* in the VCL and *TWidgetControl* in CLX) have the following events:

OnEnter *OnKeyDown* *OnKeyPress*
OnKeyUp *OnExit*

Like the standard events in *TControl*, the windowed-control events have corresponding methods. The standard key events listed above respond to all normal keystrokes.

VCL Note To respond to special keystrokes (such as the Alt key), however, you must respond to the WM_GETDLGCODE or CM_WANTSPECIALKEYS message from Windows. See Chapter 46, “Handling messages” for information on writing message handlers.

Making events visible

The declarations of the standard events in *TControl* and *TWinControl* (*TWidgetControl* in CLX) are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either public or published.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

For example, to create a component that surfaces the *OnClick* event at design time, you would add the following to the component’s class declaration.

```
type
  TMyControl = class(TCustomControl)
  ⋮
  published
    property OnClick;
  end;
```

Changing the standard event handling

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to them.

Suppose you are writing a component and you want to modify the way it responds to mouse clicks. Instead of assigning a handler to the *OnClick* event as a application developer would, you override the protected method *Click*:

```
procedure click override       { forward declaration }
⋮

procedure TMyControl.Click;
begin
  inherited Click;             { perform standard handling, including calling handler }
...                           { your customizations go here }
end;
```

Defining your own events

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

Triggering the event

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a *WM_LBUTTONDOWN* message to the application. Upon receiving that message, a component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

But some events are less clearly tied to specific external occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

Two kinds of events

There are two kinds of occurrence you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component may need to respond to. State-change events may also be related to messages from Windows (focus changes or enabling, for example), but they can also occur through changes in properties or other code.

You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.

Defining the handler type

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. All a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

Declaring the event

Once you have determined the type of your event handler, you are ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of most events in Delphi begin with "On." This is just a convention; the compiler does not enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Developers expect to find events in the alphabetical list of names starting with "On." Using other kinds of names is likely to confuse them.

Note The main exception to this rule is that many events that occur before and after some occurrence begin with "Before" and "After".

Calling the event

You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid.
- Users can override default handling.

Empty handlers must be valid

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

An empty handler should produce the same result as no handler at all. So the code for calling an application's event handler should look like this:

```
if Assigned(OnClick) then OnClick(Self);
...  { perform default handling }
```

You should *never* have something like this:

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

Users can override default handling

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

When handling key-press events, for example, application developers can suppress the component's default handling of the keystroke by setting the **var** parameter *Key* to a null character (#0). The logic for supporting this looks like

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ...  { perform default handling }
```

The actual code is a little different from this because it deals with Windows messages, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets *Key* to a null character, the component skips the default handling.

Chapter

44

Creating methods

Component methods are procedures and functions built into the structure of a class. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow. These guidelines include

- Avoiding dependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Delphi environment and are accessible at design time.

Avoiding dependencies

At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if an application calls it when the component is in a bad state, the method corrects the state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

Naming methods

Delphi imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people can use your components.

If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive. Use meaningful verbs.

  A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.

- Function names should reflect the nature of what they return.

  Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

Protecting methods

All parts of classes, including fields, methods, and properties, have a level of protection or "visibility," as explained in "Controlling access" on page 41-4. Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

Methods that should be public

Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user.

Note Constructors and destructors should always be **public**.

Methods that should be protected

Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

Abstract methods

Sometimes a method is declared as **abstract** in a Delphi component. In the VCL and CLX, abstract methods usually occur in classes whose names begin with “custom,” such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces Component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

Making methods virtual

You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

Declaring methods

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, you do two things:

- Add the declaration to the component's object-type declaration.
- Implement the method in the **implementation** part of the component's unit.

The following code shows a component that defines two new methods, one protected static method and one public virtual method.

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                               { declare protected static method }

  public
    function CalculateArea: Integer; virtual;             { declare public virtual method }
  end;
⋮
implementation
⋮
procedure TSampleComponent.MakeBigger;                           { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer;               { implement second method }
begin
  Result := Width * Height;
end;
```

Chapter

45

Using graphics in components

Windows provides a powerful Graphics Device Interface (GDI) for drawing device-independent graphics. The GDI, however, imposes extra requirements on the programmer, such as managing graphic resources. Delphi takes care of all the GDI drudgery, allowing you to focus on productive work instead of searching for lost handles or unreleased resources.

As with any part of the Windows API, you can call GDI functions directly from your Delphi application. But you will probably find that using Delphi's encapsulation of the graphic functions is faster and easier.

The topics in this section include

- Overview of graphics
- Using the canvas
- Working with pictures
- Off-screen bitmaps
- Responding to changes

Overview of graphics

Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens, brushes, and fonts. After rendering your graphic images, you must restore the device context to its original state before disposing of it.

CLX Note GDI functions are Windows-specific and do not apply to CLX or cross-platform applications.

Instead of forcing you to deal with graphics at a detailed level, Delphi provides a simple yet complete interface: your component's *Canvas* property. The canvas ensures that it has a valid device context, and releases the context when you are not using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all these resources for you, so you need not concern yourself with creating, selecting, and releasing things like pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Delphi manage graphic resources is that it can cache resources for later use, which can speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Delphi caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Delphi uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles thanks to a font cache.

Here are two examples of how simple Delphi's graphics code can be. The first uses standard GDI functions to draw a yellow ellipse outlined in blue on a window, the way you would using other development tools. The second uses a canvas to draw the same ellipse in an application written with Delphi.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  PenHandle, OldPenHandle: HPEN;
  BrushHandle, OldBrushHandle: HBRUSH;
begin
  PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));                 { create blue pen }
  OldPenHandle := SelectObject(PaintDC, PenHandle);             { tell DC to use blue pen }
  BrushHandle := CreateSolidBrush(RGB(255, 255, 0));            { create a yellow brush }
  OldBrushHandle := SelectObject(PaintDC, BrushHandle);     { tell DC to use yellow brush }
  Ellipse(HDC, 10, 10, 50, 50);                                       { draw the ellipse }
  SelectObject(OldBrushHandle);                                 { restore original brush }
  DeleteObject(BrushHandle);                                       { delete yellow brush }
  SelectObject(OldPenHandle);                                     { restore original pen }
  DeleteObject(PenHandle);                                            { destroy blue pen }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    Pen.Color := clBlue;                                               { make the pen blue }
    Brush.Color := clYellow;                                       { make the brush yellow }
    Ellipse(10, 10, 50, 50);                                            { draw the ellipse }
  end;
end;
```

Using the canvas

The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas; and in the VCL, provides low-level access to the Windows GDI.

Table 45.1 summarizes the capabilities of the canvas.

Table 45.1 Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as *MoveTo*, *LineTo*, *Rectangle*, and *Ellipse*
	Displaying and measuring text	*TextOut*, *TextHeight*, *TextWidth*, and *TextRect* methods
	Filling areas	*FillRect* and *FloodFill* methods
Intermediate	Customizing text and graphics	*Pen*, *Brush*, and *Font* properties
	Manipulating pixels	*Pixels* property.
	Copying and merging images	*Draw*, *StretchDraw*, *BrushCopy*, and *CopyRect* methods; *CopyMode* property
Low	Calling Windows GDI functions	*Handle* property

For detailed information on canvas classes and their methods and properties, see online Help.

Working with pictures

Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling stand-alone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in Delphi:

- Using a picture, graphic, or canvas
- Loading and storing graphics
- Handling palettes

Using a picture, graphic, or canvas

There are three kinds of classes in Delphi that deal with graphics:

- A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.

- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a metafile, or a user-defined graphic type, and an application can access them all in the same way through the picture class. For example, the image control has a property called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas. (The only standard graphic that has a canvas is *TBitmap*.) Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

Loading and storing graphics

All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

CLX Note You can also load images from and save them to a Qt MIME source, or a stream object if creating CLX components.

To load an image into a picture from a file, call the picture's *LoadFromFile* method. To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

To load a bitmap into an image control's picture, for example, pass the name of a bitmap file to the picture's *LoadFromFile* method:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

The picture recognizes .bmp as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

Handling palettes

For VCL components, when running on a palette-based device (typically, a 256-color video mode), Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control
- Responding to palette changes

Most controls have no need for a palette, but controls that contain “rich color” graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as *realizing* palettes.

Realizing palettes is the process of ensuring that the foremost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the “real” palette. As windows move in front of one another, Windows continually realizes the palettes.

Note Delphi itself provides no specific support for creating or maintaining palettes, other than in bitmaps. If you have a palette handle, however, Delphi controls can manage it for you.

Specifying a palette for a control

To specify a palette for a VCL control, override the control’s *GetPalette* method to return the handle of the palette.

Specifying the palette for a control does these things for your application:

- It tells the application that your control’s palette needs to be realized.
- It designates the palette to use for realization.

Responding to palette changes

If your VCL control specifies a palette by overriding *GetPalette*, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*.

The primary role of *PaletteChanged* is to determine whether to realize the control’s palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step further, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.

Off-screen bitmaps

When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in Delphi, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps.
- Copying bitmapped images.

Creating and managing off-screen bitmaps

When creating complex graphic images, you should avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an off-screen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                              { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                         { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;                                { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                             { destroy the bitmap object }
  end;
end;
```

Copying bitmapped images

Delphi provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

Table 45.2 summarizes the image-copying methods in canvas objects.

Table 45.2 Image-copying methods

To create this effect	Call this method
Copy an entire graphic.	Draw
Copy and resize a graphic.	StretchDraw
Copy part of a canvas.	CopyRect
Copy a bitmap with raster operations.	BrushCopy (VCL)
Copy a graphic repeatedly to tile an area.	TiledDraw(CLX)

Responding to changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes:

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                                { construct the pen }
  FPen.OnChange := StyleChanged;                         { assign method to OnChange event }
  FBrush := TBrush.Create;                                          { construct the brush }
  FBrush.OnChange := StyleChanged;                       { assign method to OnChange event }
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                                          { erase and repaint the component }
end;
```

Chapter

46

Handling messages

One of the keys to traditional Windows programming is handling the *messages* sent by Windows to applications. Delphi handles most of the common ones for you. It is possible, however, that you will need to handle messages that Delphi does not already handle or that you will create your own messages. CLX components do not handle Windows messages but you can create message handlers for your own messages.

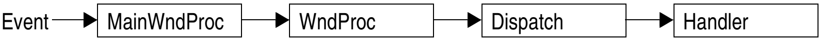
There are three aspects to working with messages:

- Understanding the message-handling system
- Changing message handling
- Creating new message handlers

Understanding the message-handling system

All Delphi classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. (Note that for CLX, the dispatch system does not include MainWndProc and WndProc.) You should never need to alter this message-dispatch mechanism. All you will need to do is create message-handling methods. See the section “Declaring a new message-handling method” on page 46-7 for more on this subject.

What's in a Windows message?

Note This information is applicable when writing VCL components only.

A Windows message is a data record that contains several fields. The most important of these is an integer-size value that identifies the message. Windows defines many messages, and the *Messages* unit declares identifiers for all of them. Other useful information in a message comes in two parameter fields and a result field.

One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as *wParam* and *lParam*, for "word parameter" and "long parameter." Often, each parameter will contain more than one piece of information, and you see references to names such as *lParamHi*, which refers to the high-order word in the long parameter.

Originally, Windows programmers had to remember or look up in the Windows API what each parameter contained. More recently, Microsoft has named the parameters. This so-called "message cracking" makes it much simpler to understand what information accompanies each message. For example, the parameters to the *WM_KEYDOWN* message are now called *nVirtKey* and *lKeyData*, which gives much more specific information than *wParam* and *lParam*.

For each type of message, Delphi defines a record type that gives a mnemonic name to each parameter. For example, mouse messages pass the x- and y-coordinates of the mouse event in the long parameter, one in the high-order word, and the other in the low-order word. Using the mouse-message structure, you do not have to worry about which word is which, because you refer to the parameters by the names *XPos* and *YPos* instead of *lParamLo* and *lParamHi*.

Dispatching messages

Note This information is applicable when writing VCL components only.

When an application creates a window, it registers a *window procedure* with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that "window" in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure.

Delphi simplifies message dispatching in several ways:

- Each component inherits a complete message-dispatching system.
- The dispatch system has default handling. You define handlers only for messages you need to respond to specially.
- You can modify small parts of the message handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component does not have a handler defined for the message, the default handling takes care of it, usually by ignoring the message.

Tracing the flow of messages

Delphi registers a method called *MainWndProc* as the window procedure for each type of component in an application. *MainWndProc* contains an exception-handling block, passing the message structure from Windows to a virtual method called *WndProc* and handling any exceptions by calling the application class's *HandleException* method.

MainWndProc is a nonvirtual method that contains no special handling for any particular messages. Customizations take place in *WndProc*, since each component type can override the method to suit its particular needs.

WndProc methods check for any special conditions that affect their processing so they can "trap" unwanted messages. For example, while being dragged, components ignore keyboard events, so the *WndProc* method of *TWinControl* passes along keyboard events only if the component is not being dragged. Ultimately, *WndProc* calls *Dispatch*, a nonvirtual method inherited from *TObject*, which determines which method to call to handle the message.

Dispatch uses the *Msg* field of the message structure to determine how to dispatch a particular message. If the component defines a handler for that particular message, *Dispatch* calls the method. If the component does not define a handler for that message, *Dispatch* calls *DefaultHandler*.

Changing message handling

Note This information is applicable when writing VCL components only.

Before changing the message handling of your components, make sure that is what you really want to do. Delphi translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change message handling in VCL components, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

Overriding the handler method

To change the way a component handles a particular message, you override the message-handling method for that message. If the component does not already handle the particular message, you need to declare a new message-handling method.

To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do *not* use the

override directive; you must use the **message** directive and a matching message index.

Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only the message index is significant. For clarity, however, it is best to follow the convention of naming message-handling methods after the messages they handle.

For example, to override a component's handling of the *WM_PAINT* message, you redeclare the *WMPaint* method:

```
type
  TMyComponent = class(...)
  ⋮
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

Using message parameters

Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the *Result* field for the message: the value returned by the *SendMessage* call that sends the message.

Note This information is applicable when writing VCL components only.

Because the type of the *Message* parameter in the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters. If for some reason you need to refer to the message parameters by their old-style names (*WParam*, *LParam*, and so on), you can typecast *Message* to the generic type *TMessage*, which uses those parameter names.

Trapping messages

Under some circumstances, you might want your components to ignore messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message, you override the virtual method *WndProc*.

For VCL components, the *WndProc* method screens messages before passing them to the *Dispatch* method, which in turn determines which method gets to handle the message. By overriding *WndProc*, your component gets a chance to filter out messages before dispatching them. An override of *WndProc* for a control derived from *TWinControl* looks like this:

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);
end;
```

The *TControl* component defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding *WndProc* helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

For CLX, a control might be descended from *TWidgetControl* and you would override *EventFilter* instead of *WndProc*.

Here is part of the *WndProc* method for *TControl*, for example:

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ⋮
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                      { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      ⋮                                                      { handle others normally }
    end;
⋮                                                       { otherwise process normally }
end;
```

Creating new message handlers

Because Delphi provides handlers for most common messages, the time you will most likely need to create new message handlers is when you define your own messages. Working with user-defined messages has two aspects:

- Defining your own messages
- Declaring a new message-handling method

CLX components do not handle Windows messages but you can create message handlers for your own messages. Note that you cannot create message handlers for Qt events because they are objects not message IDs.

Defining your own messages

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard messages and notification of state changes. You can define your own messages in both VCL and CLX.

Defining a message is a two-step process. The steps are

1. Declaring a message identifier.
2. Declaring a message-record type.

Declaring a message identifier

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant *WM_APP* represents the starting number for user-defined messages. When defining message identifiers, you should base them on *WM_APP*.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the Messages unit to see which messages Windows already defines for that control.

The following code shows two user-defined messages.

```
const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;
```

Declaring a message-record type

If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message-record is the type of the parameter passed to the message-handling method. If you do not use the message's parameters, or if you want to use the old-style parameter notation (*wParam*, *lParam*, and so on), you can use the default message-record, *TMessage*.

To declare a message-record type, follow these conventions:

1 Name the record type after the message, preceded by a *T*.

2 Call the first field in the record *Msg*, of type *TMsgParam*.

3 Define the next two bytes to correspond to the *Word* parameter, and the next two bytes as unused.

   Or

   Define the next four bytes to correspond to the *Longint* parameter.

4 Add a final field called *Result*, of type *Longint*.

For example, here is the message record for all mouse messages, *TWMMouse*, which uses a variant record to define two sets of names for the same parameters.

```
type
  TWMMouse = record
    Msg: TMsgParam;        ( first is the message ID )
    Keys: Word;            ( this is the wParam )
    case Integer of         ( two ways to look at the lParam )
      0: {
         XPos: Integer;    ( either as x- and y-coordinates...)
         YPos: Integer);
      1: {
         Pos: TPoint;      ( ... or as a single point )
         Result: Longint); ( and finally, the result field )
end;
```

Declaring a new message-handling method

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that is not already handled by the standard components.
- You have defined your own message for use by your components.

To declare a message-handling method, do the following:

1. Declare the method in a **protected** part of the component's class declaration.
2. Make the method a procedure.
3. Name the method after the message it handles, but without any underline characters.
4. Pass a single **var** parameter called *Message*, of the type of the message record.
5. Within the message method implementation, write code for any handling specific to the component.
6. Call the inherited message handler.

Here is the declaration, for example, of a message handler for a user-defined message called *CM_CHANGECOLOR*.

```
const
  CM_CHANGECOLOR = WM_APP + 400;

type
  TMyComponent = class(TControl)
  ⋮
protected
  procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;
```

Chapter

47

Making components available at design time

This chapter describes the steps for making the components you create available in the IDE. Making your components available at design time requires several steps:

- Registering components
- Adding palette bitmaps
- Providing Help for your component
- Adding property editors
- Adding component editors
- Compiling components into packages

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see "Installing component packages" on page 11-5.

Registering components

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Component palette.

Note If you create your component by choosing Component | New Component in the IDE, the code required to register your component is added automatically.

The steps for manually registering a component are:

- Declaring the Register procedure
- Writing the Register procedure

Declaring the Register procedure

Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Object Pascal) its name is case-sensitive.

The following code shows the outline of a simple unit that creates and registers new components:

```
unit MyBtns;
interface
type
  ...                                           { declare your component types here }
procedure Register;                     { this must appear in the interface section }
implementation
  ...                                          { component implementation goes here }
procedure Register;
begin
  ...                                                     { register the components }
end;
end.
```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Component palette. If the unit contains several components, you can register them all in one step.

Writing the Register procedure

Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Component palette. If the unit contains several components, you can register them at the same time.

To register a component, call the *RegisterComponents* procedure once for each page of the Component palette to which you want to add components. *RegisterComponents* involves three important things:

1. Specifying the components
2. Specifying the palette page
3. Using the RegisterComponents function

Specifying the components

Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);           { two on this page... }
  RegisterComponents('Assorted', [TThird]);                        { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the Standard page }
end;
```

Specifying the palette page

The palette-page name is a string. If the name you give for the palette page does not already exist, Delphi creates a new page with that name. Delphi stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the *LoadStr* function, passing the constant representing the string resource for that page, such as *srSystem* for the System page.

Using the RegisterComponents function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes two parameters: the name of a Component palette page and the array of component classes.

Set the Page parameter to the name of the page on the component palette where the components should appear. If the named page already exists, the components are added to that page. If the named page does not exist, Delphi creates a new palette page with that name.

Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the component palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);                 {add to system page}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);                     { new page}
end;
```

Adding palette bitmaps

Every component needs a bitmap to represent the component on the Component palette. If you don't specify your own bitmap, Delphi uses a default bitmap.

Because the palette bitmaps are needed only at design time, you don't compile them into the component's compilation unit. Instead, you supply them in a Windows resource file with the same name as the unit, but with the extension .DCR (dynamic component resource). You can create this resource file using the Image editor in Delphi. Each bitmap should be 24 pixels square.

For each component you want to install, supply a palette bitmap file, and within each palette bitmap file, supply a bitmap for each component you register. The bitmap image has the same name as the component. Keep the palette bitmap file in the same directory with the compiled files, so Delphi can find the bitmaps when it installs the components on the Component palette.

For example, if you create a component named *TMyControl* in a unit named ToolBox, you need to create a resource file called TOOLBOX.DCR that contains a bitmap called TMYCONTROL. The resource names are not case-sensitive, but by convention they are usually in uppercase letters.

Providing Help for your component

When you select a standard component on a form, or a property or event in the Object Inspector, you can press *F1* to get Help on that item. You can provide developers with the same kind of documentation for your components if you create the appropriate Help files.

You can provide a small Help file to describe your components, and your help file becomes part of the user's overall Delphi Help system.

See the section "Creating the Help file" on page 47-4 for information on how to compose the help file for use with a component.

Creating the Help file

You can use any tool you want to create the source file for a Windows Help file (in .rtf format). Delphi includes the Microsoft Help Workshop, which compiles your Help files and provides an online help authoring guide. You can find complete information about creating Help files in the online guide for Help Workshop.

Composing help files for components consists of the steps:

- Creating the entries
- Making component help context-sensitive Adding component help files

Creating the entries

To make your component's Help integrate seamlessly with the Help for the rest of the components in the library, observe the following conventions:

1 **Each component should have a help topic.**

   The component topic should show which unit the component is declared in and briefly describe the component. The component topic should link to secondary windows that describe the component's position in the object hierarchy and list all of its properties, events, and methods. Application developers access this topic by selecting the component on a form and pressing *F1*. For an example of a component topic, place any component on a form and press *F1*.

The component topic must have a # footnote with a value unique to the topic. The # footnote uniquely identifies each topic by the Help system.

The component topic should have a K footnote for keyword searching in the help system Index that includes the name of the component class. For example, the keyword footnote for the *TMemo* component is "TMemo."

The component topic should also have a $ footnote that provides the title of the topic. The title appears in the Topics Found dialog box, the Bookmark dialog box, and the History window.

2 **Each component should include the following secondary navigational topics:**

- A hierarchy topic with links to every ancestor of the component in the component hierarchy.
- A list of all properties available in the component, with links to entries describing those properties.
- A list of all events available in the component, with links to entries describing those events.
- A list of methods available in the component, with links to entries describing those methods.

Links to object classes, properties, methods, or events in the Delphi help system can be made using Alinks. When linking to an object class, the Alink uses the class name of the object, followed by an underscore and the string "object". For example, to link to the *TCustomPanel* object, use the following:

```
!AL(TCustomPanel_object,1)
```

When linking to a property, method, or event, precede the name of the property, method, or event by the name of the object that implements it and an underscore. For example, to link to the *Text* property which is implemented by *TControl*, use the following:

```
!AL(TControl_Text,1)
```

To see an example of the secondary navigation topics, display the help for any component and click on the links labeled hierarchy, properties, methods, or events.

3 **Each property, method, and event that is declared within the component should have a topic.**

A property, event, or method topic should show the declaration of the item and describe its use. Application developers see these topics either by highlighting the item in the Object Inspector and pressing *F1* or by placing the cursor in the Code editor on the name of the item and pressing *F1*. To see an example of a property topic, select any item in the Object Inspector and press *F1*.

The property, event, and method topics should include a K footnote that lists the name of the property, method, or event, and its name in combination with the name of the component. Thus, the *Text* property of *TControl* has the following K footnote:

```
Text,TControl;TControl,Text;Text,
```

The property, method, and event topics should also include a $ footnote that indicates the title of the topic, such as TControl.Text.

All of these topics should have a topic ID that is unique to the topic, entered as a # footnote.

Making component help context-sensitive

Each component, property, method, and event topic must have an A footnote. The A footnote is used to display the topic when the user selects a component and presses *F1*, or when a property or event is selected in the Object Inspector and the user presses *F1*. The A footnotes must follow certain naming conventions:

If the Help topic is for a component, the A footnote consists of two entries separated by a semicolon using this syntax:

```
ComponentClass_Object;ComponentClass
```

where *ComponentClass* is the name of the component class.

If the Help topic is for a property or event, the A footnote consists of three entries separated by semicolons using this syntax:

```
ComponentClass_Element;Element_Type;Element
```

where *ComponentClass* is the name of the component class, *Element* is the name of the property, method, or event, and *Type* is the either Property, Method, or Event

For example, for a property named *BackgroundColor* of a component named *TMyGrid*, the A footnote is

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

Adding component help files

To add your Help file to Delphi, use the OpenHelp utility (called oh.exe) located in the bin directory or accessed using Help|Customize in the IDE.

You will find information about using OpenHelp in the OpenHelp.hlp file, including adding your Help file to the Help system.

Adding property editors

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires these five steps:

1 Deriving a property-editor class

2 Editing the property as text
3 Editing the property as a whole
4 Specifying editor attributes
5 Registering the property editor

Deriving a property-editor class

Both CLX and the VCL define several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in Table 47.1. The classes in the DesignEditors unit can be used for both VCL and CLX applications. Some of the property-editor classes, however, supply specialized dialogs and so are specialized to either VCL or CLX. These can be found in the WinEditors and CLXEditors units, respectively.

Note All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface.

The list in Table 47.1 is not complete. The WinEditors and CLXEditors units also define some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

Table 47.1 Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings.
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop-down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

The following example shows the declaration of a simple property editor named *TMyPropertyEditor*:

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

Editing the property as text

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor classes provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits a set of methods used for assigning and reading different sorts of values, as shown in Table 47.2.

Table 47.2 Methods for reading and writing property values

Property type	Get method	Set method
Floating point	GetFloatValue	SetFloatValue
Method pointer (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue

When you override a *GetValue* method, you will call one of the Get methods, and when you override *SetValue*, you will call one of the Set methods.

Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns "unknown".

To provide a string representation of your property, override the property editor's *GetValue* method.

If the property is not a string value, *GetValue* must convert the value into a string representation.

Setting the property value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method.

SetValue should convert the string and validate the value before calling one of the Set methods.

Here are the *GetValue* and *SetValue* methods for *TIntegerProperty*. *Integer* is an ordinal type, so *GetValue* calls *GetOrdValue* and converts the result to a string. *SetValue* converts the string to an integer, performs some range checking, and calls *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // unsigned
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
  procedure Error(const Args: array of const);
  begin
    raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
  end;
var
  L: Int64;
begin
  L := StrToInt64(Value);
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then
    begin   // unsigned compare and reporting needed
      if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
      // bump up to Int64 to get past the %d in the format string
        Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
    end
    else if (L < MinValue) or (L > MaxValue) then
      Error([MinValue, MaxValue]);
  SetOrdValue(L);
end;
```

The specifics of the particular examples here are less important than the principle: *GetValue* converts the value to a string; *SetValue* converts the string and validates the value before calling one of the "Set" methods.

Editing the property as a whole

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

Edit methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

Within your implementation of the *Edit* method, follow these steps:

1. Construct the editor you are using for the property.
2. Read the current value and assign it to the property using a Get method.
3. When the user selects a new value, assign that value to the property using a Set method.
4. Destroy the editor.

The *Color* properties found in most components use the standard Windows color dialog box as a property editor. Here is the *Edit* method from *TColorProperty*, which invokes the dialog box and uses the result:

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);              { construct the editor }
  try
    ColorDialog.Color := GetOrdValue;                          { use the existing value }
    if ColorDialog.Execute then                          { if the user OKs the dialog... }
      SetOrdValue(ColorDialog.Color);                   { ...use the result to set value }
  finally
    ColorDialog.Free;                                              { destroy the editor }
  end;
end;
```

Specifying editor attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Table 47.3 Property-editor attribute flags

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.

Table 47.3 Property-editor attribute flags (continued)

Flag	Related method	Meaning if included
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.
paFullWidthName	N/A	The value does not need to be displayed. The Object Inspector uses its full width for the property name instead.
paVolatileSubProperties	GetProperties	The Object Inspector refetches the values of all subproperties any time the property value changes.
paReference	GetComponent Value	The value is a reference to something else. When used in conjunction with paSubProperties the referenced object should be displayed as sub properties to this property.

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

Registering the property editor

Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.

  This is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)`.

- The type of the component to which this editor applies. If this parameter is **nil**, the editor applies to all properties of the given type.

- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

The three statements in this procedure cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you have created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are **nil** and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property (of type *TComponentName*) of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

Property categories

In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit DesignIntf.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is included in the *TMiscellaneousCategory* category. Such properties are displayed or hidden in the Object Inspector based on that default categorization.

In addition to these two functions for registering properties, there is an *IsPropertyInCategory* function. This function is useful for creating localization utilities, in which you must determine whether a property is registered in a given property category.

Registering one property at a time

Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation lets you identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

The second variation is much like the first, except that it limits the category to only those properties of the given name that appear on components of a given type. The example below registers (into the 'Help and Hints' category) a property named "HelpContext" of a component of the custom class *TMyButton*.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

The third variation identifies the property using its type rather than its name. The example below registers a property based on its type, Integer.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

The final variation uses both the property's type and its name to identify the property. The example below registers a property based on a combination of its type, *TBitmap*, and its name, "Pattern".

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Registering multiple properties at once

Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name "Text" or belongs to a class of type *TEdit* is registered in the category 'Localizable'.

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

The second variation lets you limit the registered properties to those that belong to a specific component. The list of properties to register include only names, not types. For example, the following code registers a number of properties into the 'Help and Hints' category for all components:

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint',
'ParentShowHint', 'ShowHint']);
```

The third variation lets you limit the registered properties to those that have a specific type. As with the second variation, the list of properties to register can include only names:

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Specifying property categories

When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in Table 47.4.:

Table 47.4 Property categories

Category	**Purpose**
Action	Properties related to runtime actions; the *Enabled* and *Hint* properties of *TEdit* are in this category.
Database	Properties related to database operations; the *DatabaseName* and *SQL* properties of *TQuery* are in this category.
Drag, Drop, and Docking	Properties related to drag-n-drop and docking operations; the *DragCursor* and *DragKind* properties of *TImage* are in this category.
Help and Hints	Properties related to using online help or hints; the *HelpContext* and *Hint* properties of *TMemo* are in this category.
Layout	Properties related to the visual display of a control at design-time; the *Top* and *Left* properties of *TDBEdit* are in this category.
Legacy	Properties related to obsolete operations; the *Ctl3D* and *ParentCtl3D* properties of *TComboBox* are in this category.
Linkage	Properties related to associating or linking one component to another; the *DataSet* property of *TDataSource* is in this category.
Locale	Properties related to international locales; the *BiDiMode* and *ParentBiDiMode* properties of *TMainMenu* are in this category.
Localizable	Properties that may require modification in localized versions of an application. Many string properties (such as *Caption*) are in this category, as are properties that determine the size and position of controls.
Visual	Properties related to the visual display of a control at runtime; the *Align* and *Visible* properties of *TScrollBox* are in this category.
Input	Properties related to the input of data (need not be related to database operations); the *Enabled* and *ReadOnly* properties of *TEdit* are in this category.
Miscellaneous	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the AllowAllUp and Name properties of TSpeedButton are in this category.

Using the IsPropertyInCategory function

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the *IsPropertyInCategory* function are available, allowing for different criteria in determining whether a property is in a category.

The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must belong to a *TCustomEdit* descendant, have the name "Text", and be in the property category `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

The second variation lets you base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must be a *TCustomEdit* descendant, have the name "Text", and be in the property category `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', ‘TCustomEdit’, 'Text');
```

Adding component editors

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats.

If you do not give your components a component editor, Delphi uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from *TComponentEditor* and register its use with your component. In your overridden methods, you can use the *Component* property of *TComponentEditor* to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor

Adding items to the context menu

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items
- Implementing commands

Specifying menu items

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := '&DoThis ...';
    1: Result := 'Do&That';
  end;
end;
```

Note Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

Implementing commands

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous example.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
```

```
      0: begin
           MyDialog := TMySpecialDialog.Create(Application);       { instantiate the editor }
           if MySpecialDialog.Execute then;                 { if the user OKs the dialog... }
             MyComponent.FThisProperty := MySpecialDialog.ReturnValue;   { ...use the value }
           MySpecialDialog.Free;                                         { destroy the editor }
         end;
      1: That;                                                         { call the That method }
    end;
  end;
```

Changing the double-click behavior

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example, double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

Note If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of *TComponentEditor*. Then, instead of overriding the *Edit* method, override the protected *TDefaultEditor.EditProperty* method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

Adding clipboard formats

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Delphi's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Delphi IDE, but can be pasted into other applications.

```
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
```

Registering the component editor

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

Place the call to *RegisterComponentEditor* in the *Register* procedure where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit, the following code registers the component and its association with the component editor.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

Compiling components into packages

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Chapter 11, “Working with packages and components”.

To create and compile a package, see “Creating and editing packages” on page 11-6. Put the source-code units for your custom components in the package’s Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see “Installing component packages” on page 11-5.

Chapter

48

Modifying an existing component

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

Note To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template* rather than create a new class.

Modifying an existing component takes only two steps:

- Creating and registering the component
- Modifying the component class

Creating and registering the component

Creation of every component begins the same way: you create a unit, derive a component class, register it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Memos*.
- Derive a new component type called *TWrapMemo*, descended from *TMemo*.
- Register *TWrapMemo* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit Memos;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

If you compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

Modifying the component class

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor.
- Specifying the new default property value.

The constructor actually sets the value of the property. The default tells Delphi what values to store in the form (.dfm for VCL and .xfm for CLX) file. Delphi stores only values that differ from the default, so it is important to perform both steps.

Overriding the constructor

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

Note When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see "Overriding methods" on page 41-8.

For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the constructor override to the forward declaration, then write the new constructor in the **implementation** part of the unit:

```
type
  TWrapMemo = class(TMemo)
  public                                             { constructors are always public }
    constructor Create(AOwner: TComponent); override; { this syntax is always the same }
  end;
⋮
constructor TWrapMemo.Create(AOwner: TComponent);     { this goes after implementation }
begin
  inherited Create(AOwner);                          { ALWAYS do this first! }
  WordWrap := False;                                 { set the new desired value }
end;
```

Now you can install the new component on the Component palette and add it to a form. Note that the *WordWrap* property is now initialized to *False*.

If you change an initial property value, you should also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Delphi cannot store and restore the proper value.

Specifying the new default property value

When Delphi stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Chapter 47, “Making components available at design time.”

To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value. You don’t need to redeclare the entire property, just the name and the default value.

For the word-wrapping memo component, you redeclare the *WordWrap* property in the **published** part of the object declaration, with a default value of *False*:

```
type
  TWrapMemo = class(TMemo)
  ⋮
  published
    property WordWrap default False;
  end;
```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component’s constructor. Redeclaring the default ensures that Delphi knows when to write *WordWrap* to the form file.

Chapter 49

Creating a graphic component

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need a window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic component presented in this chapter is *TShape*, the shape component on the Additional page of the Component palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. This chapter calls its shape component *TSampleShape* and shows you all the steps involved in creating the shape component:

- Creating and registering the component
- Publishing inherited properties
- Adding graphic capabilities

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *Shapes*.
- Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
- Register *TSampleShape* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
```

Publishing inherited properties

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in “Publishing inherited properties” on page 42-2 and “Making events visible” on page 43-5. Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the two drag-and-drop properties:

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;       { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;       { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;      { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

Adding graphic capabilities

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control:

1 Determining what to draw.
2 Drawing the component image.

In addition, for the shape control example, you will add some properties that enable application developers to customize the appearance of the shape at design time.

Determining what to draw

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

To give your control a property that determines the shape it draws, add a property called *Shape*. This requires

1 Declaring the property type.
2 Declaring the property.
3 Writing the implementation method.

Creating properties is explained in more detail in Chapter 42, "Creating properties."

Declaring the property type

When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

Add the following type definition above the shape control class's declaration.

```
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl) { this is already there }
```

You can now use this type to declare a new property in the class.

Declaring the property

When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead.

For the shape control, you will declare a field that holds the current shape, then declare a property that reads that field and writes to it through a method call.

Add the following declarations to *TSampleShape*:

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;  { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

Now all that remains is to add the implementation of *SetShape*.

Writing the implementation method

When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method.

Add the implementation of the *SetShape* method to the **implementation** part of the unit:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                         { ignore if this isn't a change }
  begin
    FShape := Value;                              { store the new value }
    Invalidate;                                   { force a repaint with the new shape }
  end;
end;
```

Overriding the constructor and destructor

To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

Changing default property values

The default size of a graphic control is fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Chapter 48, “Modifying an existing component.”

In this example, the shape control sets its size to a square 65 pixels on each side.

Add the overridden constructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public                                                  { constructors are always public }
    constructor Create(AOwner: TComponent); override        { remember override directive }
  end;
```

1 Redeclare the *Height* and *Width* properties with their new default values:

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  published
    property Height default 65;
    property Width default 65;
  end;
```

2 Write the new constructor in the **implementation** part of the unit:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);  { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

Publishing the pen and brush

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires

1 Declaring the class fields.

2 Declaring the access properties.

3 Initializing owned classes.

4 Setting owned classes' properties.

Declaring the class fields

Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor.

Fields for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose.

Add fields for a pen and brush to the shape control:

```
type
  TSampleShape = class(TGraphicControl)
  private           { fields are nearly always private }
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush;  { a field for the brush object }
    ⋮
  end;
```

Declaring the access properties

You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the class field, but the write part calls a method that enables the component to react to changes in the owned object.

To the shape control, add properties that provide access to the pen and brush fields. You will also declare methods for reacting to changes to the pen or brush.

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  private                                        { these methods should be private }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                    { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                       { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                           { replace existing pen with parameter }
end;
```

To directly assign the contents of *Value* to *FBrush*...

```
  FBrush := Value;
```

...would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

Initializing owned classes

If you add classes to your component, the component’s constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component’s destructor must also destroy the owned objects before destroying the component itself.

Because you have added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

1 Construct the pen and brush in the shape control constructor:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                     { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                               { construct the pen }
  FBrush := TBrush.Create;                                         { construct the brush }
end;
```

2 Add the overridden destructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public                                               { destructors are always public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;                        { remember override directive }
  end;
```

3 Write the new destructor in the **implementation** part of the unit:

```
destructor TSampleShape.Destroy;
begin
  FPen.Free;                                                  { destroy the pen object }
  FBrush.Free;                                              { destroy the brush object }
  inherited Destroy;                        { always call the inherited destructor, too }
end;
```

Setting owned classes' properties

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:

```
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                     { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                               { construct the pen }
  FPen.OnChange := StyleChanged;                       { assign method to OnChange event }
  FBrush := TBrush.Create;                                        { construct the brush }
  FBrush.OnChange := StyleChanged;                      { assign method to OnChange event }
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate;                                   { erase and repaint the component }
end;
```

With these changes, the component redraws to reflect changes to either the pen or the brush.

Drawing the component image

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user.
- Use the selected shape.
- Adjust coordinates so that squares and circles use the same width and height.

Overriding the *Paint* method requires two steps:

1 Add *Paint* to the component's declaration.
2 Write the *Paint* method in the **implementation** part of the unit.

For the shape control, add the following declaration to the class declaration:

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  protected
    procedure Paint; override;
  ⋮
  end;
```

Then write the method in the **implementation** part of the unit:

```
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                              { copy the component's pen }
    Brush := FBrush;                                        { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);                 { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                                 { draw round shapes }
    end;
  end;
end;
```

Paint is called whenever the control needs to update its image. Controls are painted when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate*, as the *StyleChanged* method does.

Refining the shape drawing

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is a refined *Paint* method that adjusts for squares and ellipses:

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copy the component's pen }
    Brush := FBrush;                                    { copy the component's brush }
    W := Width;                                           { use the component width }
    H := Height;                                         { use the component height }
    if W < H then S := W else S := H;            { save smallest for circles/squares }

    case FShape of                                { adjust height, width and position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                                { origin is top-left for these shapes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                          { center these horizontally... }
          Y := (H - S) div 2;                                       { ...and vertically }
          W := S;                                { use shortest dimension for width... }
          H := S;                                                   { ...and for height }
        end;
    end;

    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);                   { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);          { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                                 { draw round shapes }
    end;
  end;
end;
```

Chapter

50

Customizing a grid

Delphi provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. In this chapter, you will see how to create a small one-month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar involves these tasks:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

The resulting component is similar to the *TCalendar* component on the Samples page of the Component palette.

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *CalSamp*.
- Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
- Register *TSampleCalendar* on the Samples page of the Component palette.

The resulting unit descending from *TCustomGrid* in the VCL should look like this:

```
unit CalSamp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;

type
  TSampleCalendar = class(TCustomGrid)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;

end.
```

If descending from the CLX version of *TCustomGrid*, only the **uses** clause would differ showing CLX units instead.

If you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in “Filling in the cells” below.

Publishing inherited properties

The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control.

To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component’s declaration.

For the calendar control, publish the following properties and events, as shown here:

```
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align;  { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;  { publish events }
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;
```

There are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you install the modified calendar component to the Component palette and use it in an application, you will find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

Changing initial values

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual.

Remember that you need to add the constructor to the **public** part of the component's object declaration, then write the new constructor in the **implementation** part of the component's unit. The first statement in the new constructor should always be a call to the inherited constructor.

```
type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner: TComponent); override;
  ⋮
  end;
⋮
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                { call inherited constructor }
  ColCount := 7;                                             { always seven days/week }
  RowCount := 7;                                { always six weeks plus the headings }
  FixedCols := 0;                                                    { no row labels }
  FixedRows := 1;                                              { one row for day names }
  ScrollBars := ssNone;                                            { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];  {disable range selection}
end;
```

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

Resizing the cells

VCL When a user or application changes the size of a window or control, Windows sends a message called *WM_SIZE* to the affected window or control so it can adjust any settings needed to later paint its image in the new size. Your VCL component can respond to that message by altering the size of the cells so they all fit inside the boundaries of the control. To respond to the *WM_SIZE* message, you will add a message-handling method to the component.

Creating a message-handling method is described in detail in "Creating new message handlers" on page 46-5.

In this case, the calendar control needs a response to *WM_SIZE*, so add a protected method called *WMSize* to the control indexed to the *WM_SIZE* message, then write the method so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  ⋮
  end;
⋮
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;                                        { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                 { calculate combined size of all lines }
  DefaultColWidth := (Message.Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Message.Height - GridLines) div 7;             { and cell height }
end;
```

Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

CLX In CLX, changes to the size of a window or control are automatically notified by a call to the protected *BoundsChanged* method. Your CLX component can respond to this notification by altering the size of the cells so they all fit inside the boundaries of the control.

In this case, the calendar control needs to override *BoundsChanged* so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  ⋮
  end;
```

```
⋮
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;                                    { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                  { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7;            { and cell height }
  inherited; {now call the inherited method }
end;
```

Filling in the cells

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
  end;
⋮
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);    { use RTL strings }
end;
```

Tracking the date

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. Delphi stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date

- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day

Storing the internal date

To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.

Adding the internal date to the calendar requires three steps:

1 Declare a private field to hold the date:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  ⋮
```

2 Initialize the date field in the constructor:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);         { this is already here }
  ⋮                                 { other initializations here }
  FDate := Date;                    { get current date from RTL }
end;
```

3 Declare a runtime property to allow access to the encoded date.

You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ⋮
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                 { set new date value }
  Refresh;                        { update the onscreen image }
end;
```

Accessing the day, month, and year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

To provide design-time access to the day, month, and year, you do the following:

1 Declare the three properties, assigning each a unique **index** number:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ⋮
```

2 Declare and write the implementation methods, setting different elements for each index value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;        { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);             { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                     { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                 { get current date elements }
    case Index of                                     { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);                 { encode the modified date }
    Refresh;                                                { update the visible calendar }
  end;
end;
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at runtime using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

Generating the day numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you will need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in a class field, then update that field each time the date changes.

To fill in the days in the proper cells, you do the following:

1 Add a month-offset field to the object and a method that updates the field value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                     { storage for the offset }
  ⋮
  protected
    procedure UpdateCalendar; virtual;                      { property for offset access }
  end;
⋮
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                                { date of the first day of the month }
begin
  if FDate <> 0 then                                 { only calculate offset if date is valid }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                          { get elements of date }
    FirstDate := EncodeDate(AYear, AMonth, 1);                          { date of the first }
    FMonthOffset := 2 - DayOfWeek(FirstDate);           { generate the offset into the grid }
  end;
  Refresh;                                                        { always repaint the control }
end;
```

2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                         { this is already here }
  ⋮                                                          { other initializations here }
  UpdateCalendar;                                                      { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
```

```
begin
  FDate := Value;                                              { this was already here }
  UpdateCalendar;                                        { this previously called Refresh }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  ⋮
    FDate := EncodeDate(AYear, AMonth, ADay);                 { encode the modified date }
    UpdateCalendar;                                      { this previously called Refresh }
  end;
end;
```

3 Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;          { calculate day for this cell }
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                              { return -1 if invalid }
end;
```

Remember to add the declaration of *DayNum* to the component's type declaration.

4 Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                        { if this is the header row ...}
    TheText := ShortDayNames[ACol + 1]                          { just use the day name }
  else begin
    TheText := '';                                          { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                          { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay);       { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
```

Now if you reinstall the calendar component and place one on a form, you will see the proper information for the current month.

Selecting the current day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    ⋮ { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
```

Note that you are now reusing the *ADay* variable previously set by decoding the date.

Navigating months and years

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a “next month” feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year. After incrementing or decrementing *CalendarDate*, decode the date value to fill the Year, Month, and Day properties with corresponding new values.

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
  DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
end;
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

Navigating days

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

The process of navigating days consists of

- Moving the selection
- Providing an OnChange event
- Excluding blank cells

Moving the selection

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*, including the **override** directive afterward.

```
procedure TSampleCalendar.Click;
var
  TempDay: Integer;
begin
  inherited Click;                             { remember to call the inherited method! }
  TempDay := DayNum(Col, Row);                { get the day number for the clicked cell }
  if TempDay <> -1 then Day := TempDay;                            { change day if valid }
end;
```

Providing an OnChange event

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

Add an *OnChange* event to *TSampleCalendar*.

1 Declare the event, a field to store the event, and a dynamic method to call the event:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  ⋮
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  ⋮
```

2 Write the *Change* method:

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                               { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ⋮                                           { many statements setting element values }
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                                  { this is new }
  end;
end;
```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

Excluding blank cells

As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells.

To control whether a given cell is selectable, override the *SelectCell* method of the grid.

SelectCell is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

You can override *SelectCell* to return *False* if the cell does not contain a valid date:

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False           { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);      { otherwise, use inherited value }
end;
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.

Chapter

51

Making a control data aware

When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. Delphi includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Chapter 15, “Using data controls”.

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This chapter first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the *TSampleCalendar* calendar created in Chapter 50, “Customizing a grid”. You can also use the standard calendar control on the Samples page of the Component palette, *TCalendar*.

The chapter then continues with an explanation of how to make the new data-browsing control a data-editing control.

Creating a data-browsing control

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component
- Adding the data link
- Responding to data changes

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *DBCal*.
- Derive a new component class called *TDBCalendar*, descended from the VCL component *TSampleCalendar*. Chapter 50, “Customizing a grid,” shows you how to create the *TSampleCalendar* component.
- Register *TDBCalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit DBCal;

interface

uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Grids, Calendar;

type
  TDBCalendar = class(TSampleCalendar)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;

end.
```

You can now proceed with making the new calendar a data browser.

Making the control read-only

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves,

- Adding the ReadOnly property.
- Allowing needed updates.

Note that if you started with the *TCalendar* component from Delphi’s Samples page instead of *TSampleCalendar*, it already has a *ReadOnly* property, so you can skip these steps.

Adding the ReadOnly property

By adding a *ReadOnly* property, you will provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unselectable.

1 Add the property declaration and a **private** field to hold the value:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;                                   { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;    { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                       { always call the inherited constructor! }
  FReadOnly := True;                                                { set the default value }
end;
```

2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in “Excluding blank cells” on page 50-12.

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                       { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCalendar*, and append the **override** directive.

If you now add the calendar to a form, you will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

Allowing needed updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                  { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
```

```
    procedure UpdateCalendar; override;              { remember the override directive }
  end;
  ⋮
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False      { allow select if updating }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;

procedure TDBCalendar.UpdateCalendar;
begin
  FUpdating := True;                                         { set flag to allow updates }
  try
    inherited UpdateCalendar;                                          { update as usual }
  finally
    FUpdating := False;                                            { always clear the flag }
  end;
end;
```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data-browsing ability.

Adding the data link

The connection between a control and a database is handled by a class called a *data link*. The datalink class that connects a control with a single field in a database is *TFieldDataLink* (VCL or CLX). There are also data links for entire tables.

A data-aware control *owns* its datalink class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Chapter 49, “Creating a graphic component”.

Establishing a data link as an owned class requires these three steps:

1 Declaring the class field

2 Declaring the access properties

3 Initializing the data link

Declaring the class field

A component needs a field for each of its owned classes, as explained in “Declaring the class fields” on page 49-5. In this case, the calendar needs a field of type *TFieldDataLink* for its data link.

Declare a field for the data link in the calendar:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
  ⋮
  end;
```

Before you can compile the application, you need to add DB and DBCtrls to the unit's **uses** clause.

Declaring the access properties

Every data-aware control has a *DataSource* property that specifies which data-source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes in the example in Chapter 49, "Creating a graphic component", these access properties do not provide access to the owned classes themselves, but rather to corresponding properties in the owned class. That is, you will create properties that enable the control and its data link to share the same data source and field.

Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as "pass-through" methods to the corresponding properties of the datalink class:

An example of declaring access properties

```
type
  TDBCalendar = class(TSampleCalendar)
  private                                          { implementation methods are private }
    ...
    function GetDataField: string;                 { returns the name of the data field }
    function GetDataSource: TDataSource;         { returns reference to the data source }
    procedure SetDataField(const Value: string);           { assigns name of data field }
    procedure SetDataSource(Value: TDataSource);             { assigns new data source }
  published                                 { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
⋮
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
```

Now that you have established the links between the calendar and its data link, there is one more important step. You must construct the data link class when the calendar control is constructed, and destroy the data link before destroying the calendar.

Initializing the data link

A data-aware control needs access to its data link throughout its existence, so it must construct the datalink object as part of its own constructor, and destroy the datalink object before it is itself destroyed.

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink object, respectively:

```
type
  TDBCalendar = class(TSampleCalendar)
  public                                 { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                 { always call the inherited constructor first
}
  FDataLink := TFieldDataLink.Create;                    { construct the datalink object }
  FDataLink.Control := self;               {let the datalink know about the calendar }
  FReadOnly := True;                                               { this is already here }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;                                 { always destroy owned objects first... }
  inherited Destroy;                                  { ...then call inherited destructor
}
end;
```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section explains how to do that.

Responding to data changes

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Datalink classes all have events named *OnDataChange*. When the data source indicates a change in its data, the datalink object calls any event handler attached to its *OnDataChange* event.

To update a control in response to data changes, attach a handler to the data link’s *OnDataChange* event.

In this case, you will add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnDataChange* event in the constructor. In the destructor, detach the *OnDataChange* handler before destroying the object.

```
type
  TDBCalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject);         { must have proper parameters for event
}
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                  { always call the inherited constructor first
}
  FReadOnly := True;                                               { this is already here }
  FDataLink := TFieldDataLink.Create;                    { construct the datalink object }
  FDataLink.OnDataChange := DataChange;                         { attach handler to event }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;                 { detach handler before destroying object }
  FDataLink.Free;                                  { always destroy owned objects first... }
  inherited Destroy;                                   { ...then call inherited destructor
}
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then                         { if there is no field assigned...
}
    CalendarDate := 0                                            { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDateTime;   { otherwise, set calendar to the date }
end;
```

You now have a data-browsing control.

Creating a data-editing control

When you create a data-editing control, you create and register the component and add the data link just as you do for a data-browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data-editing control described here is the same calendar control described in the first part of the chapter. The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data-editing control involves:

- Changing the default value of FReadOnly.
- Handling mouse-down and key-down messages.
- Updating the field datalink class.
- Modifying the Change method.
- Updating the dataset.

Changing the default value of FReadOnly

Because this is a data-editing control, the *ReadOnly* property should be set to *False* by default. To make the *ReadOnly* property *False*, change the value of *FReadOnly* in the constructor:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  ⋮
  FReadOnly := False;  { set the default value }
  ⋮
end;
```

Handling mouse-down and key-down messages

When the user of the control begins interacting with it, the control receives either mouse-down messages (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN*, or *WM_RBUTTONDOWN*) or a key-down message (*WM_KEYDOWN*) from Windows. If using CLX, notification is from the operating system in the form of system events. To enable a control to respond to these messages, you must write handlers that respond to these messages.

- Responding to mouse-down messages
- Responding to key-down messages

Responding to mouse-down messages

A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a Windows mouse-down message. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event.

To override *MouseDown*, add the *MouseDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
   ⋮
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
   ⋮
  end;
```

```
procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is *False* and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

Responding to key-down messages

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override *KeyDown*, follow these steps:

1 Add a *KeyDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
   ⋮
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
   ⋮
  end;
```

2 Implement the *KeyDown* method:

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
```

When *KeyDown* responds to a mouse-down message, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is *False*, the key pressed is one of the cursor control keys, and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

Updating the field datalink class

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field datalink class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

1. To reflect a change made to the value in the calendar in the field value, add an *UpdateData* method to the private section of the calendar component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
   ⋮
  end;
```

2. Implement the *UpdateData* method:

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;        { set field link to calendar date }
end;
```

3. Within the constructor for *TDBCalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

Modifying the Change method

The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code. These are the steps to follow:

1 Add a new *Change* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
   ⋮
  end;
```

2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;                  { call the Modified method }
  inherited Change;                    { call the inherited Change method }
end;
```

Updating the dataset

So far, a change within the data-aware control has changed values in the field datalink class. The final step in creating a data-editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the *Tab* key. This process works differently in the VCL and CLX.

VCL VCL has defined message control IDs for operations on controls. For example, the *CM_EXIT* message is sent to the control when the user exits the control. You can write message handlers that respond to the message. In this case, when the user exits the control, the *CMExit* method, the message handler for *CM_EXIT*, responds by updating the record in the dataset with the changed values in the field datalink class. For more information about message handlers, see Chapter 46, “Handling messages.”

To update the dataset within a message handler, follow these steps:

1 Add the message handler to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
   ⋮
  end;
```

2 Implement the *CMExit* method so it looks like this:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update database }
  except
    on Exception do SetFocus;                      { if it failed, don't let focus leave }
 end;
  inherited;
end;
```

CLX In CLX, *TWidgetControl* has a protected *DoExit* method that is called when input focus shifts away from the control. This method calls the event handler for the *OnExit* event. You can override this method to update the record in the dataset before generating the *OnExit* event handler.

To update the dataset when the user exits the control, follow these steps:

1 Add an override for the *DoExit* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
    ⋮
  end;
```

2 Implement the *DoExit* method so it looks like this:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update database }
  except
    on Exception do SetFocus;                      { if it failed, don't let focus leave }
 end;
  inherited;                        { let the inherited method generate an OnExit event }

end;
```

Chapter

52

Making a dialog box a component

You will find it convenient to make a frequently used dialog box into a component that you add to the Component palette. Your dialog box components will work just like the components that represent the standard common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time.

Making a dialog box a component requires these steps:

1. Defining the component interface
2. Creating and registering the component
3. Creating the component interface
4. Testing the component

The Delphi "wrapper" component associated with the dialog box creates and executes the dialog box at runtime, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this chapter, you will see how to create a wrapper component around the generic About Box form provided in the Delphi Object Repository.

Note Copy the files ABOUT.PAS and ABOUT.DFM into your working directory.

There are not many special considerations for designing a dialog box that will be wrapped into a component. Nearly any form can operate as a dialog box in this context.

Defining the component interface

Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it.

For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and initial control settings, then read back any needed information after the dialog box closes. There is no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component.

The interface must therefore contain enough information that the dialog box form can appear in the way the developer specifies and return any information the application needs. You can think of the properties in the wrapper component as being persistent data for a transient dialog box.

In the case of the About box, you do not need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Because there are four separate fields in the About box that the application might affect, you will provide four string-type properties to provide for them.

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the component palette. This process is outlined in "Creating a new component" on page 40-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *AboutDlg*.
- Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*.
- Register *TAboutBoxDlg* on the Samples page of the component palette.

The resulting unit should look like this:

```
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

The new component now has only the capabilities built into *TComponent*. It is the simplest nonvisual component. In the next section, you will create the interface between the component and the dialog box.

Creating the component interface

These are the steps to create the component interface:

1. Including the form unit
2. Adding interface properties
3. Adding the Execute method

Including the form unit

For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the **uses** clause of the wrapper component's unit.

Append *About* to the **uses** clause of the *AboutDlg* unit.

The **uses** clause now looks like this:

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,
  About;
```

The form unit always declares an instance of the form class. In the case of the About box, the form class is *TAboutBox*, and the *About* unit includes the following declaration:

```
var
  AboutBox: TAboutBox;
```

So by adding *About* to the **uses** clause, you make *AboutBox* available to the wrapper component.

Adding interface properties

Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's class declaration.

Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you are just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set data at design time so that the wrapper can pass it to the dialog box at runtime.

Declaring an interface property requires two additions to the component's class declaration:

- A private class field, which is a variable the wrapper uses to store the value of the property
- The published property declaration itself, which specifies the name of the property and tells it which field to use for storage

Interface properties of this sort do not need access methods. They use direct access to their stored data. By convention, the class field that stores the property's value has the same name as the property, but with the letter *F* in front. The field and the property *must* be of the same type.

For example, to declare an integer-type interface property called *Year*, you would declare it as follows:

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;                                  { field to hold the Year-property data }
  published
    property Year: Integer read FYear write FYear;        { property matched with storage }
  end;
```

For this About box, you need four string-type properties—one each for the product name, the version information, the copyright information, and any comments.

```
type
  TAboutBoxDlg = class(TComponent)
  private
    FProductName, FVersion, FCopyright, FComments: string;               { declare fields }

  published
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

When you install the component onto the component palette and place the component on a form, you will be able to set the properties, and those values will automatically stay with the form. The wrapper can then use those values when executing the wrapped dialog box.

Adding the Execute method

The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common-dialog-box components, you will use a boolean function called *Execute* that returns *True* if the user clicks OK, or *False* if the user cancels the dialog box.

The declaration for the *Execute* method always looks like this:

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

The minimum implementation for *Execute* needs to construct the dialog box form, show it as a modal dialog box, and return either *True* or *False*, depending on the return value from *ShowModal*.

Here is the minimal *Execute* method for a dialog-box form of type *TMyDialogBox*:

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);                    { construct the form }
  try
    Result := (DialogBox.ShowModal = IDOK);     { execute; set result based on how closed }
  finally
    DialogBox.Free;                                                  { dispose of the form }
  end;
end;
```

Note the use of a **try..finally** block to ensure that the application disposes of the dialog box object even if an exception occurs. In general, whenever you construct an object this way, you should use a **try..finally** block to protect the block of code and make certain the application frees any resources it allocates.

In practice, there will be more code inside the **try..finally** block. Specifically, before calling *ShowModal*, the wrapper will set some of the dialog box's properties based on the wrapper component's interface properties. After *ShowModal* returns, the wrapper will probably set some of its interface properties based on the outcome of the dialog box execution.

In the case of the About box, you need to use the wrapper component's four interface properties to set the contents of the labels in the About box form. Because the About box does not return any information to the application, there is no need to do anything after calling *ShowModal*. Write the About box wrapper's *Execute* method so that it looks like this:

Within the public part of the *TAboutDlg* class, add the declaration for the *Execute* method:

```
type
  TAboutDlg = class(TComponent)
public
  function Execute: Boolean;
end;

function TAboutBoxDlg.Execute: Boolean;
begin
  AboutBox := TAboutBox.Create(Application);                       { construct About box }
  try
    if ProductName = '' then                              { if product name's left blank... }
      ProductName := Application.Title;                  { ...use application title instead }
    AboutBox.ProductName.Caption := ProductName;                         { copy product name }
    AboutBox.Version.Caption := Version;                                 { copy version info }
    AboutBox.Copyright.Caption := Copyright;                           { copy copyright info }
    AboutBox.Comments.Caption := Comments;                                   { copy comments }
    AboutBox.Caption := 'About ' + ProductName;                      { set About-box caption }
    with AboutBox do begin
      ProgramIcon.Picture.Graphic := Application.Icon;                           { copy icon }
      Result := (ShowModal = IDOK);                                { execute and set result }
    end;
  finally
    AboutBox.Free;                                                    { dispose of About box }
  end;
end;
```

Testing the component

Once you have installed the dialog-box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

For example, if you created an About dialog box, made it a component, and added it to the Component palette, you can test it with the following steps:

1 Create a new project.

2 Place an About-box component on the main form.

3 Place a command button on the form.

4 Double-click the command button to create an empty click-event handler.

5 In the click-event handler, type the following line of code:

```
AboutBoxDlg1.Execute;
```

6 Run the application.

When the main form appears, click the command button. The About box appears with the default project icon and the name Project1. Choose OK to close the dialog box.

You can further test the component by setting the various properties of the About box component and again running the application.

Index

Object Pascal Language Guide

Borland®

Object Pascal

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Printed in the U.S.A.

ALP0000WW21000 1E0R0501
0102030405-9 8 7 6 5 4 3 2 1
D3

Contents

Chapter 5
Data types, variables, and constants 5-1

Chapter 6
Procedures and functions 6-1

Chapter 7
Classes and objects 7-1

Part II
Special topics

Chapter 11
Memory management 11-1

Chapter 12
Program control 12-1

Chapter 13
Inline assembler code 13-1

Appendix A
Object Pascal grammar A-1

Index I-1

Tables

Chapter

1

Introduction

This manual describes the Object Pascal programming language as it is used in Borland development tools.

What's in this manual?

The first seven chapters describe most of the language elements used in ordinary programming. Chapter 8 summarizes standard routines for file I/O and string manipulation.

The next chapters describe language extensions and restrictions for dynamic-link libraries and packages (Chapter 9), and for object interfaces (Chapter 10). The final three chapters address advanced topics: memory management (Chapter 11), program control (Chapter 12), and assembly-language routines within Object Pascal programs (Chapter 13).

Using Object Pascal

The *Object Pascal Language Guide* is written to describe the Object Pascal language for use on either the Linux or Windows operating systems. Differences in the language relating to platform dependencies are noted where necessary.

Most Delphi/Kylix application developers write and compile their Object Pascal code in the integrated development environment (IDE). Working in the IDE allows the product to handle many details of setting up projects and source files, such as maintenance of dependency information among units. Borland products may enforce certain constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, certain file- and program-naming conventions can be avoided if you write your programs outside of the IDE and compile them from the command prompt.

This manual generally assumes that you are working in the IDE and that you are building applications that use the Visual Component Library (VCL) and/or the Borland Component Library for Cross Platform (CLX). Occasionally, however, Borland-specific rules are distinguished from rules that apply to all Object Pascal programming.

Typographical conventions

Identifiers—that is, names of constants, variables, types, fields, properties, procedures, functions, programs, units, libraries, and packages—appear in *italics* in the text. Object Pascal operators, reserved words, and directives are in **boldface** type. Example code and text that you would type literally (into a file or at the command prompt) are in `monospaced` type.

In displayed program listings, reserved words and directives appear in **boldface**, just as they do in the text:

```
function Calculate(X, Y: Integer): Integer;
begin
  ⋮
end;
```

This is how the Code editor displays reserved words and directives, if you have the Syntax Highlight option turned on.

Some program listings, like the example above, contain ellipsis marks (... or ⋮). The ellipses represent additional code that would be included in an actual file. They are not meant to be copied literally.

In syntax descriptions, *italics* indicate placeholders for which, in real code, you would substitute syntactically valid constructions. For example, the heading of the function declaration above could be represented as

`function` *functionName*`(`*argumentList*`):` *returnType*`;`

Syntax descriptions can also contain ellipsis marks (...) and subscripts:

`function` *functionName*`(`arg_1`, ...,` arg_n`):` *ReturnType*`;`

Other sources of information

The online Help system for your development tool provides information about the IDE and user interface as well as the most up-to-date reference material for the VCL and/or CLX. Many programming topics, such as database development, are covered in depth in the *Developer's Guide*. For an overview of the documentation set, see the *Quick Start* manual that came with your software package.

Software registration and technical support

Borland Software Corporation offers a range of support plans to fit the needs of individual developers, consultants, and corporations. To receive help with this product, return the registration card and select the plan that best suits your needs. For additional information about technical support and other Borland services, contact your local sales representative or visit us online at http://www.borland.com/.

Part

I

Basic language description

The chapters in Part I present the essential language elements required for most programming tasks. These chapters include:

- Chapter 2, “Overview”
- Chapter 3, “Programs and units”
- Chapter 4, “Syntactic elements”
- Chapter 5, “Data types, variables, and constants”
- Chapter 6, “Procedures and functions”
- Chapter 7, “Classes and objects”
- Chapter 8, “Standard routines and I/O”

Chapter

2

Overview

Object Pascal is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming.

Object Pascal has special features that support Borland's component framework and RAD environment. For the most part, descriptions and examples in this manual assume that you are using Object Pascal to develop applications using Borland development tools such as Delphi or Kylix.

Program organization

Programs are usually divided into source-code modules called *units*. Each program begins with a heading, which specifies a name for the program. The heading is followed by an optional **uses** clause, then a block of declarations and statements. The **uses** clause lists units that are linked into the program; these units, which can be shared by different programs, often have **uses** clauses of their own.

The **uses** clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, Object Pascal programs do not require makefiles, header files, or preprocessor "include" directives. (The Project Manager generates a makefile each time a project is loaded in the IDE, but saves these files only for project groups that include more than one project.)

For further discussion of program structure and dependencies, see Chapter 3, "Programs and units".

Pascal source files

The compiler expects to find Pascal source code in files of three kinds:

- *unit* source files (which end with the .pas extension)

- *project* files (which end with the .dpr extension)
- *package* source files (which end with the .dpk extension)

Unit source files contain most of the code in an application. Each application has a single project file and several unit files; the project file—which corresponds to the "main" program file in traditional Pascal—organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. But if you use the IDE to build your application, you must have a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called *packages*. For more information about packages, see Chapter 9, "Libraries and packages".

Other files used to build applications

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically and include

- *form* files, which end with the .dfm (Delphi) or .xfm (Kylix) extension,
- *resource* files, which end with the .res extension, and
- *project options* files, which end with the .dof (Delphi) or .kof (Kylix) extension.

A form file is either a text file or a compiled resource file that can contain bitmaps, strings, and so forth. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to form files, each project uses a resource (.res) file to hold the bitmap for the application's icon. By default, this file has the same name as the project (.dpr) file. To change an application's icon, use the Project Options dialog.

A project options (.dof or .kof) file contains compiler and linker settings, search directories, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk or .desk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

Compiler-generated files

The first time you build an application or a standard dynamic-link library, the compiler produces a compiled unit .dcu (Windows) .dcu/.dpu (Linux) file for each new unit used in your project; all the .dcu (Windows) .dcu/.dpu (Linux) files in your project are then linked to create a single executable or shared library file. The first time you build a package, the compiler produces a .dcu (Windows) .dpu (Linux) file for each new unit contained in the package, and then creates both a .dcp and a package file. (For more information about libraries and packages, see Chapter 9.) If you use the **–GD** switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you rebuild a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, or their .dcu (Windows) .dcu/ .dpu (Linux) files cannot be found, or you explicitly tell the compiler to reprocess them. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file.

Example programs

The examples that follow illustrate basic features of Object Pascal programming. The examples show simple Object Pascal applications that cannot be compiled from the IDE; but you can compile them from the command line.

A simple console application

The program below is a simple console application that you can compile and run from the command prompt.

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
  MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

The first line declares a program called *Greeting*. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called *MyMessage*, which holds a string. (Object Pascal has genuine string data types.) The program then assigns the string "Hello world!" to the variable *MyMessage*, and sends the contents of *MyMessage* to the standard output using the *Writeln* procedure. (*Writeln* is defined implicitly in the *System* unit, which the compiler automatically includes in every application.)

You can type this program into a file called Greeting.pas or Greeting.dpr and compile it by entering

```
On Delphi: DCC32 Greeting

On Kylix: dcc Greeting
```

on the command line. The resulting executable prints the message "Hello world!"

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Borland development tools. First, it is a console application. Borland development tools are typically used to write applications with graphical interfaces; hence, you would not ordinarily call *Writeln*. Moreover, the entire example program (save for *Writeln*) is in a single file. In a typical application, the program heading—the first line of the example—would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to *methods* defined in unit files.

A more complicated example

The next example shows a program that is divided into two files: a *project* file and a *unit* file. The project file, which you can save as Greeting.dpr, looks like this:

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
  PrintMessage('Hello World!');
end.
```

The first line declares a program called *Greeting*, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that *Greeting* includes a unit called *Unit1*. Finally, the program calls the *PrintMessage* procedure, passing to it the string "Hello World!" Where does the *PrintMessage* procedure come from? It's defined in *Unit1*. Here's the source code for *Unit1*, which you can save in a file called Unit1.pas:

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
  Writeln(msg);
end;

end.
```

Unit1 defines a procedure called *PrintMessage* that takes a single string as an argument and sends the string to the standard output. (In Pascal, routines that do not return a value are called *procedures*. Routines that return a value are called *functions*.) Notice that *PrintMessage* is declared twice in *Unit1*. The first declaration, under the reserved word **interface**, makes *PrintMessage* available to other modules (such as *Greeting*) that use *Unit1*. The second declaration, under the reserved word **implementation**, actually defines *PrintMessage*.

You can now compile *Greeting* from the command line by entering

```
On Delphi: DCC32 Greeting
```

```
On Kylix: dcc Greeting
```

There's no need to include *Unit1* as a command-line argument. When the compiler processes Greeting.dpr, it automatically looks for unit files that the *Greeting* program depends on. The resulting executable does the same thing as our first example: it prints the message "Hello world!"

A native application

Our next example is an application built using VCL or CLX components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of Object Pascal. In addition to multiple units, the program uses classes and objects, which are discussed in Chapter 7, "Classes and objects".

The program includes a project file and two new unit files. First, the project file:

```
program Greeting;  { comments are enclosed in braces }

uses
  Forms, {change the unit name to QForms on Linux}
  Unit1 in 'Unit1.pas' { the unit for Form1 },
  Unit2 in 'Unit2.pas' { the unit for Form2 };

{$R *.res}  { this directive links the project's resource file }

begin
  { calls to Application }
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Once again, our program is called *Greeting*. It uses three units: *Forms*, which is part of VCL and CLX; *Unit1*, which is associated with the application's main form (*Form1*); and *Unit2*, which is associated with another form (*Form2*).

The program makes a series of calls to an object named *Application*, which is an instance of the *TApplication* class defined in the *Forms* unit. (Every project has an automatically generated *Application* object.) Two of these calls invoke a *TApplication* method named *CreateForm*. The first call to *CreateForm* creates *Form1,* an instance of the *TForm1* class defined in *Unit1*. The second call to *CreateForm* creates *Form2,* an instance of the *TForm2* class defined in *Unit2*.

Unit1 looks like this:

```
unit Unit1;

interface

uses  { these units are part of the Visual Component Library (VCL) }
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

{
On Linux, the uses clause looks like this:
uses  { these units are part of CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

uses Unit2;  { this is where Form2 is defined }

{$R *.dfm}  { this directive links Unit1's form file }

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.
```

Unit1 creates a class named *TForm1* (derived from *TForm*) and an instance of this class*, Form1*. *TForm1* includes a button—*Button1,* an instance of *TButton*—and a procedure named *TForm1.Button1Click* that is called at runtime whenever the user presses *Button1*. *TForm1.Button1Click* hides *Form1* and it displays *Form2* (the call to *Form2.ShowModal*). *Form2* is defined in *Unit2*:

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
{
On Linux, the uses clause looks like this:
uses  { these units are part of CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
  TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
  Form2.Close;
end;

end.
```

Unit2 creates a class named *TForm2* and an instance of this class, *Form2*. *TForm2* includes a button (*CancelButton*, an instance of *TButton*) and a label (*Label1*, an instance of *TLabel*). You can't see this from the source code, but *Label1* displays a caption that reads "Hello world!" The caption is defined in *Form2*'s form file, Unit2.dfm.

Unit2 defines one procedure. *TForm2.CancelButtonClick* is called at runtime whenever the user presses *CancelButton*; it closes *Form2*. This procedure (along with *Unit1*'s *TForm1.Button1Click*) is known as an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form (.dfm on Windows .xfm on Linux) files for *Form1* and *Form2*.

When the *Greeting* program starts, *Form1* is displayed and *Form2* is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's *main form*.) When the user presses the button on *Form1*, *Form1* disappears and is replaced by *Form2*, which displays the "Hello world!" greeting. When the user closes *Form2* (by pressing *CancelButton* or the Close button on the title bar), *Form1* reappears.

Chapter

3

Programs and units

A program is constructed from source-code modules called *units*. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. Units allow you to

- divide large programs into modules that can be edited separately.
- create libraries that you can share among programs.
- distribute libraries to other developers without making the source code available.

In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use a *project* (.dpr) file to store the "main" program, while most other source code resides in *unit* (.pas) files. Each application—or *project*—consists of a single project file and one or more unit files. (Strictly speaking, you needn't explicitly use any units in a project, but all programs automatically use the *System* unit.) To build a project, the compiler needs either a source file or a compiled unit file for each unit.

Program structure and syntax

A program contains

- a program heading,
- a **uses** clause (optional), and
- a block of declarations and statements.

The program heading specifies a name for the program. The **uses** clause lists units used by the program. The block contains declarations and statements that are executed when the program runs. The IDE expects to find these three elements in a single project (.dpr) file.

The example below shows the project file for a program called Editor.

```
 1    program Editor;
 2
 3    uses
 4      Forms, {change to QForms in Linux}
 5      REAbout in 'REAbout.pas' {AboutBox},
 6      REMain in 'REMain.pas' {MainForm};
 7
 8    {$R *.res}
 9
10    begin
11      Application.Title := 'Text Editor';
12      Application.CreateForm(TMainForm, MainForm);
13      Application.Run;
14    end.
```

Line 1 contains the program heading. The **uses** clause is on lines 3 through 6. Line 8 is a compiler directive that links the project's resource file into the program. Lines 10 through 14 contain the block of statements that are executed when the program runs. Finally, the project file, like all source files, ends with a period.

This is, in fact, a fairly typical project file. Project files are usually short, since most of a program's logic resides in its unit files. Project files are generated and maintained automatically, and it is seldom necessary to edit them manually.

The program heading

The program heading specifies the program's name. It consists of the reserved word **program**, followed by a valid identifier, followed by a semicolon. The identifier must match the project file name. In the example above, since the program is called Editor, the project file should be called EDITOR.dpr.

In standard Pascal, a program heading can include parameters after the program name:

```
program Calc(input, output);
```

Borland's Object Pascal compiler ignores these parameters.

The program uses clause

The **uses** clause lists units that are incorporated into the program. These units may in turn have **uses** clauses of their own. For more information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

The block

The block contains a simple or structured statement that is executed when the program runs. In most programs, the block consists of a compound statement—bracketed between the reserved words **begin** and **end**—whose component statements are simply method calls to the project's *Application* object. (Every project has an *Application* variable that holds an instance of *TApplication*, *TWebApplication*, or *TServiceApplication*.) The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

Unit structure and syntax

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own unit (.pas) file.

A unit file begins with a unit heading, which is followed by the *interface*, *implementation*, *initialization*, and *finalization* sections. The initialization and finalization sections are optional. A skeleton unit file looks like this:

```
unit Unit1;

interface

uses  { List of units goes here }

  { Interface section goes here }

implementation

uses  { List of units goes here }

  { Implementation section goes here }

initialization
  { Initialization section goes here }

finalization
  { Finalization section goes here }

end.
```

The unit must conclude with the word **end** followed by a period.

The unit heading

The unit heading specifies the unit's name. It consists of the reserved word **unit**, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the unit file name. Thus, the unit heading

```
unit MainForm;
```

would occur in a source file called MAINFORM.pas, and the file containing the compiled unit would be MAINFORM.dcu.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

The interface section

The interface section of a unit begins with the reserved word **interface** and continues until the beginning of the implementation section. The interface section declares constants, types, variables, procedures, and functions that are available to *clients*—that is, to other units or programs that use the unit where they are declared. These entities are called *public* because a client can access them as if they were declared in the client itself.

The interface declaration of a procedure or function includes only the routine's heading. The block of the procedure or function follows in the implementation section. Thus procedure and function declarations in the interface section work like forward declarations, although the **forward** directive isn't used.

The interface declaration for a class must include declarations for all class members.

The interface section can include its own **uses** clause, which must appear immediately after the word **interface**. For information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

The implementation section

The implementation section of a unit begins with the reserved word **implementation** and continues until the beginning of the initialization section or, if there is no initialization section, until the end of the unit. The implementation section defines procedures and functions that are declared in the interface section. Within the implementation section, these procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the implementation section; but if you include a parameter list, it must match the declaration in the interface section exactly.

In addition to definitions of public procedures and functions, the implementation section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit—that is, inaccessible to clients.

The implementation section can include its own **uses** clause, which must appear immediately after the word **implementation**. For information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

The initialization section

The initialization section is optional. It begins with the reserved word **initialization** and continues until the beginning of the finalization section or, if there is no finalization section, until the end of the unit. The initialization section contains statements that are executed, in the order in which they appear, on program start-up.

So, for example, if you have defined data structures that need to be initialized, you can do this in the initialization section.

The initialization sections of units used by a client are executed in the order in which the units appear in the client's **uses** clause.

The finalization section

The finalization section is optional and can appear only in units that have an initialization section. The finalization section begins with the reserved word **finalization** and continues until the end of the unit. It contains statements that are executed when the main program terminates. Use the finalization section to free resources that are allocated in the initialization section.

Finalization sections are executed in the opposite order from initializations. For example, if your application initializes units *A*, *B*, and *C*, in that order, it will finalize them in the order *C*, *B*, and *A*.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the initialization code might not execute completely.

Unit references and the uses clause

A **uses** clause lists units used by the program, library, or unit in which the clause appears. (For information about libraries, see Chapter 9, "Libraries and packages".) A **uses** clause can occur in

- the project file for a program or library,
- the interface section of a unit, and
- the implementation section of a unit.

Most project files contain a **uses** clause, as do the interface sections of most units. The implementation section of a unit can contain its own **uses** clause as well.

The *System* unit is used automatically by every application and cannot be listed explicitly in the **uses** clause. (*System* implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as *SysUtils*, must be included in the **uses** clause. In most cases, all necessary units are placed in the **uses** clause when your project generates and maintains a source file.

For more information about the placement and content of the **uses** clause, see "Multiple and indirect unit references" on page 3-6 and "Circular unit references" on page 3-7.

The syntax of a uses clause

A **uses** clause consists of the reserved word **uses**, followed by one or more comma-delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;

uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

In the **uses** clause of a program or library, any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;

uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

Include **in** ... after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no reason to do this. Using **in** is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path or the general Library search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the **in** ... construction to determine which units are part of a project. Only units that appear in a project (.dpr) file's **uses** clause followed by **in** and a file name are considered to be part of the project; other units in the **uses** clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the Project Manager and Project Browser.

In the **uses** clause of a unit, you cannot use **in** to tell the compiler where to find a source file. Every unit must be in the compiler's search path, the general Library search path, or the same directory as the unit that uses it. Moreover, unit names must match the names of their source files.

Multiple and indirect unit references

The order in which units appear in the **uses** clause determines the order of their initialization (see "The initialization section" on page 3-4) and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the same name, the compiler uses the one from the unit listed last in the **uses** clause. (To access the identifier from the other unit, you would have to add a qualifier: *UnitName.Identifier*.)

A **uses** clause need include only units used directly by the program or unit in which the clause appears. That is, if unit *A* references constants, types, variables, procedures, or functions that are declared in unit *B*, then *A* must use *B* explicitly. If *B* in turn references identifiers from unit *C*, then *A* is indirectly dependent on *C*; in this case, *C* needn't be included in a **uses** clause in *A*, but the compiler must still be able to find both *B* and *C* in order to process *A*.

The example below illustrates indirect dependency.

```
program Prog;
uses Unit2;
const a = b;
⋮

unit Unit2;
interface
uses Unit1;
const b = c;
⋮

unit Unit1;
interface
const c = 1;
⋮
```

In this example, *Prog* depends directly on *Unit2*, which depends directly on *Unit1*. Hence *Prog* is indirectly dependent on *Unit1*. Because *Unit1* does not appear in *Prog*'s **uses** clause, identifiers declared in *Unit1* are not available to *Prog*.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their .dcu (Windows) or .dcu/.dpu (Linux) files, not their source (.pas) files.

When changes are made in the interface section of a unit, other units that depend on it must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

Circular unit references

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the **uses** clause of one interface section to the **uses** clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the **uses** clause of at least one implementation section.

In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface **uses** clauses. So the following example leads to a compilation error:

```
unit Unit1;
interface
uses Unit2;
⋮

unit Unit2;
interface
uses Unit1;
⋮
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;
interface
uses Unit2;
⋮

unit Unit2;
interface
⋮
implementation
uses Unit1;
⋮
```

To reduce the chance of circular references, it's a good idea to list units in the implementation **uses** clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface **uses** clause.

Chapter

4

Syntactic elements

Object Pascal uses the ASCII character set, including the letters *A* through *Z* and *a* through *z*, the digits *0* through *9*, and other standard characters. It is *not* case-sensitive. The space character (ASCII 32) and the control characters (ASCII 0 through 31—including ASCII 13, the return or end-of-line character) are called *blanks*.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

Fundamental syntactic elements

On the simplest level, a program is a sequence of tokens delimited by separators. A *token* is the smallest meaningful unit of text in a program. A *separator* is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;
Price := 10;
```

Tokens are categorized as *special symbols*, *identifiers*, *reserved words*, *directives*, *numerals*, *labels*, and *character strings*. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

Special symbols

Special symbols are nonalphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols.

# $ & ' ( ) * + , – . / : ; < = > @ [ ] ^ { }

The following character pairs are also special symbols.

(* (. *) .) .. // := <= >= <>

The left bracket—**[**—is equivalent to the character pair of left parenthesis and period—**(.**; the right bracket—**]**—is equivalent to the character pair of period and right parenthesis—**.)** . The left-parenthesis–plus–asterisk and asterisk–plus–right-parenthesis—**(* *)**—are equivalent to the left and right brace—**{ }**.

Notice that **!**, **"** (double quotation marks), **%**, **?**, ****, **_** (underscore), **|** (pipe), and **~** (tilde) are not special characters.

Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with a letter or an underscore (_) and cannot contain spaces; letters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Since Object Pascal is case-insensitive, an identifier like *CalculateValue* could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

On Linux, the only identifiers for which case is important are unit names. Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation.

Qualified identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to *qualify* the identifier. The syntax for a qualified identifier is

identifier$_1$.*identifier*$_2$

where *identifier*$_1$ qualifies *identifier*$_2$. For example, if two units each declare a variable called *CurrentValue*, you can specify that you want to access the *CurrentValue* in *Unit2* by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the *Click* method in *Button1* of *Form1*.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in "Blocks and scope" on page 4-27.

Reserved words

The following reserved words cannot be redefined or used as identifiers.

Table 4.1 Reserved words

and	**downto**	**in**	**or**	**string**
array	**else**	**inherited**	**out**	**then**
as	**end**	**initialization**	**packed**	**threadvar**
asm	**except**	**inline**	**procedure**	**to**
begin	**exports**	**interface**	**program**	**try**
case	**file**	**is**	**property**	**type**
class	**finalization**	**label**	**raise**	**unit**
const	**finally**	**library**	**record**	**until**
constructor	**for**	**mod**	**repeat**	**uses**
destructor	**function**	**nil**	**resourcestring**	**var**
dispinterface	**goto**	**not**	**set**	**while**
div	**if**	**object**	**shl**	**with**
do	**implementation**	**of**	**shr**	**xor**

In addition to the words in Table 4.1, **private**, **protected**, **public**, **published**, and **automated** act as reserved words within object type declarations, but are otherwise treated as directives. The words **at** and **on** also have special meanings.

Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in Object Pascal, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence—although it is inadvisable to do so—you can define an identifier that looks exactly like a directive.

Table 4.2 Directives

absolute	**dynamic**	**message**	**private**	**resident**
abstract	**export**	**name**	**protected**	**safecall**
assembler	**external**	**near**	**public**	**stdcall**
automated	**far**	**nodefault**	**published**	**stored**
cdecl	**forward**	**overload**	**read**	**varargs**
contains	**implements**	**override**	**readonly**	**virtual**
default	**index**	**package**	**register**	**write**
deprecated	**library**	**pascal**	**reintroduce**	**writeonly**
dispid	**local**	**platform**	**requires**	

Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the **+** or **–** operator to indicate sign. Values default to positive (so that, for example, `67258` is equivalent to `+67258`) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character *E* or *e* occurs within a real, it means “times ten to the power of”. For example, `7E-2` means 7×10^{-2}, and `12.25e+6` and `12.25e6` both mean 12.25×10^{6}.

The dollar-sign prefix indicates a hexadecimal numeral—for example, `$8F`. For the Integer type (16-bit integer), the sign of a hexadecimal is determined by the leftmost (most significant) bit of its binary representation. For all other types, you must use a prefixed + or - operator to indicate sign.

For more information about real and integer types, see Chapter 5, “Data types, variables, and constants”. For information about the data types of numerals, see “True constants” on page 5-39.

Labels

A label is a sequence of no more than four digits—that is, a numeral between 0 and 9999. Leading zeros are not significant. Identifiers can also function as labels.

Labels are used in **goto** statements. For more information about **goto** statements and labels, see “Goto statements” on page 4-18.

Character strings

A character string, also called a *string literal* or *string constant*, consists of a *quoted string*, a *control string*, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND'          { BORLAND }
'You''ll see'      { You'll see }
''''               { ' }
''                 { null string }
' '                { a space }
```

A control string is a sequence of one or more *control characters*, each of which consists of the **#** symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-return–line-feed between “Line 1” and “Line 2”. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the + operator described in “String operators” on page 4-9, or simply combine them into a single quoted string.)

A character string’s *length* is the number of characters in the string. A character string of any length is compatible with any string type and with the *PChar* type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (**{$X+}**), a character string of length *n* ≥ 1 is compatible with zero-based arrays and packed arrays of *n* characters. For more information about string types, see Chapter 5, “Data types, variables, and constants”.

Comments and compiler directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }

(* Text between a left-parenthesis-plus-asterisk and an
   asterisk-plus-right-parenthesis also constitutes a comment. *)

// Any text between a double-slash and the end of the line constitutes a comment.
```

A comment that contains a dollar sign (**$**) immediately after the opening **{** or **(*** is a compiler directive. For example,

```
{$WARNINGS OFF}
```

tells the compiler not to generate warning messages.

Expressions

An *expression* is a construction that returns a value. For example,

```
X                    { variable }
@X                   { address of a variable }
15                   { integer constant }
InterestRate         { variable }
Calc(X,Y)            { function call }
X * Y                { product of X and Y }
Z / (1 - Z)          { quotient of Z and (1 - Z) }
X = 1.5              { Boolean }
C in Range1          { Boolean }
not Done             { negation of a Boolean }
```

```
['a','b','c']        { set }
Char(48)             { value typecast }
```

The simplest expressions are variables and constants (described in Chapter 5, "Data types, variables, and constants"). More complex expressions are built from simpler ones using *operators*, *function calls*, *set constructors*, *indexes*, and *typecasts*.

Operators

Operators behave like predefined functions that are part of the Object Pascal language. For example, the expression `(X + Y)` is built from the variables `X` and `Y`—called *operands*—with the **+** operator; when `X` and `Y` represent integers or reals, `(X + Y)` returns their sum. Operators include **@**, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **–**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in**, and **is**.

The operators **@**, **not**, and **^** are *unary* (taking one operand). All other operators are *binary* (taking two operands), except that **+** and **–** can function as either unary or binary. A unary operator always precedes its operand (for example, `-B`), except for **^**, which follows its operand (for example, `P^`). A binary operator is placed between its operands (for example, `A = 7`).

Some operators behave differently depending on the type of data passed to them. For example, **not** performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories.

Except for **^**, **is**, and **in**, all operators can take operands of type *Variant*. For details, see "Variant types" on page 5-30.

The sections that follow assume some familiarity with Object Pascal data types. For information about data types, see Chapter 5, "Data types, variables, and constants".

For information about operator precedence in complex expressions, see "Operator precedence rules" on page 4-12.

Arithmetic operators

Arithmetic operators, which take real or integer operands, include **+**, **–**, *****, **/**, **div**, and **mod**.

Table 4.3 Binary arithmetic operators

Operator	Operation	Operand types	Result type	Example
+	addition	integer, real	integer, real	`X + Y`
–	subtraction	integer, real	integer, real	`Result - 1`
*****	multiplication	integer, real	integer, real	`P * InterestRate`
/	real division	integer, real	real	`X / 2`
div	integer division	integer	integer	`Total` **div** `UnitSize`
mod	remainder	integer	integer	`Y` **mod** `6`

Table 4.4 Unary arithmetic operators

Operator	Operation	Operand type	Result type	Example
+	sign identity	integer, real	integer, real	`+7`
–	sign negation	integer, real	integer, real	`-X`

The following rules apply to arithmetic operators.

- The value of *x*/*y* is of type *Extended*, regardless of the types of *x* and *y*. For other arithmetic operators, the result is of type *Extended* whenever at least one operand is a real; otherwise, the result is of type *Int64* when at least one operand is of type *Int64*; otherwise, the result is of type *Integer*. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of *x* **div** *y* is the value of *x*/*y* rounded in the direction of zero to the nearest integer.
- The **mod** operator returns the remainder obtained by dividing its operands. In other words, *x* **mod** *y* = *x* – (*x* **div** *y*) * *y*.
- A runtime error occurs when *y* is zero in an expression of the form *x*/*y*, *x* **div** *y*, or *x* **mod** *y*.

Boolean operators

The Boolean operators **not**, **and**, **or**, and **xor** take operands of any Boolean type and return a value of type *Boolean*.

Table 4.5 Boolean operators

Operator	Operation	Operand types	Result type	Example
not	negation	Boolean	*Boolean*	**not** (C **in** MySet)
and	conjunction	Boolean	*Boolean*	Done **and** (Total > 0)
or	disjunction	Boolean	*Boolean*	A **or** B
xor	exclusive disjunction	Boolean	*Boolean*	A **xor** B

These operations are governed by standard rules of Boolean logic. For example, an expression of the form *x* **and** *y* is *True* if and only if both *x* and *y* are *True*.

Complete versus short-circuit Boolean evaluation

The compiler supports two modes of evaluation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation. *Complete evaluation* means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. *Short-circuit evaluation* means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when *A* is *False*, the compiler won't evaluate *B*; it knows that the entire expression is *False* as soon as it evaluates *A*.

Short-circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string *S*, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  ⋮
  Inc(I);
end;
```

In a case where *S* has no commas, the last iteration increments *I* to a value which is greater than the length of *S*. When the **while** condition is next tested, complete evaluation results in an attempt to read *S[I]*, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the **while** condition—`(S[I] <> ',')`—is not evaluated after the first part fails.

Use the **$B** compiler directive to control evaluation mode. The default state is **{$B–}**, which enables short-circuit evaluation. To enable complete evaluation locally, add the **{$B+}** directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting Complete Boolean Evaluation in the Compiler Options dialog.

Note If either operand involves a variant, the compiler always performs complete evaluation (even in the **{$B–}** state).

Logical (bitwise) operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in *X* (in binary) is 001101 and the value stored in *Y* is 100001, the statement

```
Z := X or Y;
```

assigns the value 101101 to *Z*.

Table 4.6 Logical (bitwise) operators

Operator	Operation	Operand types	Result type	Examples
not	bitwise negation	integer	integer	**not** X
and	bitwise and	integer	integer	X **and** Y
or	bitwise or	integer	integer	X **or** Y
xor	bitwise xor	integer	integer	X **xor** Y
shl	bitwise shift left	integer	integer	X **shl** 2
shr	bitwise shift right	integer	integer	Y **shr** I

The following rules apply to bitwise operators.

- The result of a **not** operation is of the same type as the operand.

- If the operands of an **and**, **or**, or **xor** operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations *x* **shl** *y* and *x* **shr** *y* shift the value of *x* to the left or right by *y* bits, which is equivalent to multiplying or dividing *x* by 2^y; the result is of the same type as *x*. For example, if *N* stores the value 01101 (decimal 13), then `N shl 1` returns 11010 (decimal 26). Note that the value of y is interpreted modulo the size of the type of x. Thus for example, if x is an integer, x shl 40 is interpreted as x shl 8 because an integer is 32 bits and 40 mod 32 is 8.

String operators

The relational operators **=**, **<>**, **<**, **>**, **<=**, and **>=** all take string operands (see "Relational operators" on page 4-10). The **+** operator concatenates two strings.

Table 4.7 String operators

Operator	Operation	Operand types	Result type	Example
+	concatenation	string, packed string, character	string	`S + '. '`

The following rules apply to string concatenation.

- The operands for **+** can be strings, packed strings (packed arrays of type *Char*), or characters. However, if one operand is of type *WideChar*, the other operand must be a long string.
- The result of a **+** operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

Pointer operators

The relational operators **<**, **>**, **<=**, and **>=** can take operands of type *PChar* (see "Relational operators" on page 4-10). The following operators also take pointers as operands. For more information about pointers, see "Pointers and pointer types" on page 5-25.

Table 4.8 Character-pointer operators

Operator	Operation	Operand types	Result type	Example
+	pointer addition	character pointer, integer	character pointer	`P + I`
-	pointer subtraction	character pointer, integer	character pointer, integer	`P - Q`
^	pointer dereference	pointer	base type of pointer	`P^`
=	equality	pointer	*Boolean*	`P = Q`
<>	inequality	pointer	*Boolean*	`P <> Q`

The **^** operator dereferences a pointer. Its operand can be a pointer of any type except the generic *Pointer*, which must be typecast before dereferencing.

P **=** *Q* is *True* just in case *P* and *Q* point to the same address; otherwise, *P* **<>** *Q* is *True*.

You can use the **+** and **–** operators to increment and decrement the offset of a character pointer. You can also use **–** to calculate the difference between the offsets of two character pointers. The following rules apply.

- If *I* is an integer and *P* is a character pointer, then *P* **+** *I* adds *I* to the address given by *P*; that is, it returns a pointer to the address *I* characters after *P*. (The expression *I* **+** *P* is equivalent to *P* **+** *I*.) *P* **–** *I* subtracts *I* from the address given by *P*; that is, it returns a pointer to the address *I* characters before *P*.
- If *P* and *Q* are both character pointers, then *P* **–** *Q* computes the difference between the address given by *P* (the higher address) and the address given by *Q* (the lower address); that is, it returns an integer denoting the number of characters between *P* and *Q*. *P* **+** *Q* is not defined.

Set operators

The following operators take sets as operands.

Table 4.9 Set operators

Operator	Operation	Operand types	Result type	Example
+	union	set	set	`Set1 + Set2`
–	difference	set	set	`S - T`
*	intersection	set	set	`S * T`
<=	subset	set	*Boolean*	`Q <= MySet`
>=	superset	set	*Boolean*	`S1 >= S2`
=	equality	set	*Boolean*	`S2 = MySet`
<>	inequality	set	*Boolean*	`MySet <> S1`
in	membership	ordinal, set	*Boolean*	`A` **in** `Set1`

The following rules apply to **+**, **–**, and *****.

- An ordinal *O* is in *X* **+** *Y* if and only if *O* is in *X* or *Y* (or both). *O* is in *X* **–** *Y* if and only if *O* is in *X* but not in *Y*. *O* is in *X* ***** *Y* if and only if *O* is in both *X* and *Y*.
- The result of a **+**, **–**, or ***** operation is of the type **set of** *A*..*B*, where *A* is the smallest ordinal value in the result set and *B* is the largest.

The following rules apply to **<=**, **>=**, **=**, **<>**, and **in**.

- *X* **<=** *Y* is *True* just in case every member of *X* is a member of *Y*; *Z* **>=** *W* is equivalent to *W* **<=** *Z*. *U* **=** *V* is *True* just in case *U* and *V* contain exactly the same members; otherwise, *U* **<>** *V* is *True*.
- For an ordinal *O* and a set *S*, *O* **in** *S* is *True* just in case *O* is a member of *S*.

Relational operators

Relational operators are used to compare two operands. The operators **=**, **<>**, **<=**, and **>=** also apply to sets (see “Set operators” on page 4-10); **=** and **<>** also apply to pointers (see “Pointer operators” on page 4-9).

Table 4.10 Relational operators

Operator	Operation	Operand types	Result type	Example
=	equality	simple, class, class reference, interface, string, packed string	*Boolean*	`I = Max`
<>	inequality	simple, class, class reference, interface, string, packed string	*Boolean*	`X <> Y`
<	less-than	simple, string, packed string, *PChar*	*Boolean*	`X < Y`
>	greater-than	simple, string, packed string, *PChar*	*Boolean*	`Len > 0`
<=	less-than-or-equal-to	simple, string, packed string, *PChar*	*Boolean*	`Cnt <= I`
>=	greater-than-or-equal-to	simple, string, packed string, *PChar*	*Boolean*	`I >= 1`

For most simple types, comparison is straightforward. For example, *I* = *J* is *True* just in case *I* and *J* have the same value, and *I* <> *J* is *True* otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordering of the extended ASCII character set. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with *n* components is compared to a string, the packed string is treated as a string of length *n*.
- The operators <, >, <=, and >= apply to *PChar* operands only if the two pointers point within the same character array.
- The operators = and <> can take operands of class and class-reference types. With operands of a class type, = and <> are evaluated according the rules that apply to pointers: *C* = *D* is *True* just in case *C* and *D* point to the same instance object, and *C* <> *D* is *True* otherwise. With operands of a class-reference type, *C* = *D* is *True* just in case *C* and *D* denote the same class, and *C* <> *D* is *True* otherwise. For more information about classes, see Chapter 7, “Classes and objects”.

Class operators

The operators **as** and **is** take classes and instance objects as operands; **as** operates on interfaces as well. For more information, see Chapter 7, “Classes and objects” and Chapter 10, “Object interfaces”.

The relational operators = and <> also operate on classes. See “Relational operators” on page 4-10.

The @ operator

The @ operator returns the address of a variable, or of a function, procedure, or method; that is, @ constructs a pointer to its operand. For more information about pointers, see "Pointers and pointer types" on page 5-25. The following rules apply to @.

- If *X* is a variable, @*X* returns the address of *X*. (Special rules apply when *X* is a procedural variable; see "Procedural types in statements and expressions" on page 5-29.) The type of @*X* is *Pointer* if the default **{$T–}** compiler directive is in effect. In the **{$T+}** state, @*X* is of type **^***T*, where *T* is the type of *X*.
- If *F* is a routine (a function or procedure), @*F* returns *F*'s entry point. The type of @*F* is always *Pointer*.
- When @ is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

  ```
  @TMyClass.DoSomething
  ```

  points to the *DoSomething* method of *TMyClass*. For more information about classes and methods, see Chapter 7, "Classes and objects".

Operator precedence rules

In complex expressions, rules of precedence determine the order in which operations are performed.

Table 4.11 Precedence of operators

Operators	Precedence
@, **not**	first (highest)
*, /, **div**, **mod**, **and**, **shl**, **shr**, **as**	second
+, –, **or**, **xor**	third
=, <>, <, >, <=, >=, **in**, **is**	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

```
X + Y * Z
```

multiplies *Y* times *Z*, then adds *X* to the result; * is performed first, because is has a higher precedence than **+**. But

```
X - Y + Z
```

first subtracts *Y* from *X*, then adds *Z* to the result; **–** and **+** have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

```
(X + Y) * Z
```

multiplies *Z* times the sum of *X* and *Y*.

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

```
X = Y or X = Z
```

The intended interpretation of this is obviously

```
(X = Y) or (X = Z)
```

Without parentheses, however, the compiler follows operator precedence rules and reads it as

```
(X = (Y or X)) = Z
```

—which results in a compilation error unless *Z* is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example above could be written as

```
X + (Y * Z)
```

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

Function calls

Because functions return a value, function calls are expressions. For example, if you've defined a function called *Calc* that takes two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If *I* and *J* are integer variables, then `I + Calc(J, 8)` is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

For more information about functions, see Chapter 6, "Procedures and functions".

Set constructors

A set constructor denotes a set-type value. For example,

```
[5, 6, 7, 8]
```

denotes the set whose members are 5, 6, 7, and 8. The set constructor

```
[ 5..8 ]
```

could also denote the same set.

The syntax for a set constructor is

[$item_1$, ..., $item_n$]

where each *item* is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (**..**) in between. When an *item* has the form *x*..*y*, it is shorthand for all the ordinals in the range from *x* to *y*, inclusive; but if *x* is greater than *y*, then *x*..*y* denotes nothing and **[***x*..*y***]** is the empty set. The set constructor `[ ]` denotes the empty set, while **[***x***]** denotes the set whose only member is the value of *x*.

Examples of set constructors:

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see "Sets" on page 5-17.

Indexes

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if *FileName* is a string variable, the expression `FileName[3]` returns the third character in the string denoted by *FileName*, while `FileName[I + 1]` returns the character immediately after the one indexed by *I*. For information about strings, see "String types" on page 5-10. For information about arrays and array properties, see "Arrays" on page 5-18 and "Array properties" on page 7-19.

Typecasts

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, `Integer('A')` casts the character *A* as an integer.

The syntax for a typecast is

typeIdentifier(*expression*)

If the expression is a variable, the result is called a *variable typecast*; otherwise, the result is a *value typecast*. While their syntax is the same, different rules apply to the two kinds of typecast.

Value typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal types or both be pointer types. Examples of value typecasts include

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

The statement

```
I := Integer('A');
```

assigns the value of `Integer('A')`—that is, 65—to the variable *I*.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

Variable typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like *Int* and *Trunc*.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
⋮
Shortint(MyChar) := 122;
```

assigns the character *z* (ASCII 122) to *MyChar*.

You can cast variables to a procedural type. For example, given the declarations

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can make the following assignments.

```
F := Func(P);          { Assign procedural value in P to F }
Func(P) := F;          { Assign procedural value in F to P }
@F := P;               { Assign pointer value in P to F }
P := @F;               { Assign pointer value in F to P }
N := F(N);             { Call function via F }
N := Func(P)(N);       { Call function via P }
```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
  end;
  PByte = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;
begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

In this example, *TByteRec* is used to access the low- and high-order bytes of a word, and *TWordRec* to access the low- and high-order words of a long integer. You could call the predefined functions *Lo* and *Hi* for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see “Pointers and pointer types” on page 5-25. For information about casting class and interface types, see “The as operator” on page 7-25 and “Interface typecasts” on page 10-10.

Declarations and statements

Aside from the **uses** clause (and reserved words like **implementation** that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

Declarations

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like `26057` are not identifiers.) Identifiers must be *declared* before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable *Result* when it occurs inside a function block, and the variable *Self* when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called *Size* that holds an *Extended* (real) value, while

```
function DoThis(X, Y: string): Integer;
```

declares a function called *DoThis* that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
  Size: Extended;
  Quantity: Integer;
  Description: string;
```

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or implementation section of a unit (after the **uses** clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the chapters on those topics.

The "hint" directives **platform**, **deprecated**, and **library** may be appended to any declaration, except that units cannot be declared with **deprecated**. In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall; deprecated;

var VersionNumber: Real library;

type AppError = class(Exception)
  ⋮
end platform;
```

When source code is compiled in the **{$HINTS ON} {$WARNINGS ON}** state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use **platform** to mark items that are specific to a particular operating environment (such as Windows or Linux), **deprecated** to indicate that an item is obsolete or supported only for backward compatibility, and **library** to flag dependencies on a particular library or component framework (such as VCL or CLX).

Statements

Statements define algorithmic actions within a program. Simple statements—like assignments and procedure calls—can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

Simple statements

A simple statement doesn't contain any other statements. Simple statements include assignments, calls to procedures and functions, and **goto** jumps.

Assignment statements

An assignment statement has the form

variable := *expression*

where *variable* is any variable reference—including a variable, variable typecast, dereferenced pointer, or component of a structured variable—and *expression* is any assignment-compatible expression. (Within a function block, *variable* can be replaced with the name of the function being defined. See Chapter 6, "Procedures and functions".) The **:=** symbol is sometimes called the *assignment operator*.

An assignment statement replaces the current value of *variable* with the value of *expression*. For example,

```
I := 3;
```

assigns the value 3 to the variable *I*. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

```
I := I + 1;
```

increments the value of *I*. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Procedure and function calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

With extended syntax enabled (**{$X+}**), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see Chapter 6, “Procedures and functions”.

Goto statements

A **goto** statement, which has the form

goto *label*

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

label: *statement*

Declare labels like this:

label *label*;

You can declare several labels at once:

```
label label1, ..., labeln;
```

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and **goto** statement must belong to the same block. (See "Blocks and scope" on page 4-27.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;
⋮
StartHere: Beep;
goto StartHere;
```

creates an infinite loop that calls the *Beep* procedure repeatedly.

The **goto** statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

  ⋮ {code to execute if no answer is found }
  Exit;

  FoundAnAnswer:
    ⋮ { code to execute when an answer is found }
end;
```

Notice that we are using **goto** to jump *out* of a nested loop. Never jump *into* a loop or other structured statement, since this can have unpredictable effects.

Structured statements

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or **with** statement simply executes a sequence of constituent statements.
- A conditional statement—that is, an **if** or **case** statement—executes at most one of its constituents, depending on specified criteria.

- Loop statements—including **repeat**, **while**, and **for** loops—execute a sequence of constituent statements repeatedly.
- A special group of statements—including **raise**, **try...except**, and **try...finally** constructions—create and handle *exceptions*. For information about exception generation and handling, see "Exceptions" on page 7-26.

Compound statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words **begin** and **end**, and its constituent statements are separated by semicolons. For example:

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

The last semicolon before **end** is optional. So we could have written this as

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

Compound statements are essential in contexts where Object Pascal syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
  I := SomeConstant;
  while I > 0 do
  begin
    ⋮
    I := I - 1;
  end;
end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, **begin** and **end** sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

With statements

A **with** statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a **with** statement is

with *obj* do *statement*

or

with obj_1, ..., obj_n do *statement* where *obj* is a variable reference denoting an object or record, and *statement* is any simple or structured statement. Within *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone—without qualifiers.

For example, given the declarations

```
type TDate = record
  Day: Integer;
  Month: Integer;
  Year: Integer;
end;

var OrderDate: TDate;
```

you could write the following **with** statement.

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1;
  end
  else
    Month := Month + 1;
```

This is equivalent to

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1;
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before *statement* is executed. This makes **with** statements efficient as well as concise. It also means that assignments to a variable within *statement* cannot affect the interpretation of *obj* during the current execution of the **with** statement.

Each variable reference or method name in a **with** statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the **with** statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year
    ⋮
  end;
```

When multiple objects or records appear after **with**, the entire statement is treated like a series of nested **with** statements. Thus

with obj_1, obj_2, ..., obj_n do *statement* is equivalent to

```
with obj1 do
  with obj2 do
     ⋮
     with objn do
       statement
```

In this case, each variable reference or method name in *statement* is interpreted, if possible, as a member of obj_n; otherwise it is interpreted, if possible, as a member of obj_{n-1}; and so forth. The same rule applies to interpreting the *obj*s themselves, so that, for instance, if obj_n is a member of both obj_1 and obj_2, it is interpreted as $obj_2.obj_n$.

If statements

There are two forms of **if** statement: **if...then** and the **if...then...else**. The syntax of an **if...then** statement is

```
if expression then statement
```

where *expression* returns a Boolean value. If *expression* is *True*, then *statement* is executed; otherwise it is not. For example,

```
if J <> 0 then Result := I/J;
```

The syntax of an **if...then...else** statement is

```
if expression then statement1 else statement2
```

where *expression* returns a Boolean value. If *expression* is *True*, then $statement_1$ is executed; otherwise $statement_2$ is executed. For example,

```
if J = 0 then
  Exit
else
  Result := I/J;
```

The **then** and **else** clauses contain one statement each, but it can be a structured statement. For example,

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True
else
  Exit;
```

Notice that there is never a semicolon between the **then** clause and the word **else**. You can place a semicolon after an entire **if** statement to separate it from the next statement in its block, but the **then** and **else** clauses require nothing more than a space or carriage return between them. Placing a semicolon immediately before **else** (in an **if** statement) is a common programming error.

A special difficulty arises in connection with nested **if** statements. The problem arises because some **if** statements have **else** clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer **else** clauses than **if** statements, it may not seem clear which **else** clauses are bound to which **if**s. Consider a statement of the form

if *expression*$_1$ then if *expression*$_2$ then *statement*$_1$ else *statement*$_2$;

There would appear to be two ways to parse this:

if *expression*$_1$ then [if *expression*$_2$ then *statement*$_1$ else *statement*$_2$];

if *expression*$_1$ then [if *expression*$_2$ then *statement*$_1$] else *statement*$_2$;

The compiler always parses in the first way. That is, in real code, the statement

```
if ... { expression1 } then
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 } ;
```

is equivalent to

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 }
end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each **else** bound to the nearest available **if** on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { statement1 }
end
else
  ... { statement2 } ;
```

Case statements

The **case** statement provides a readable alternative to complex nested **if** conditionals. A **case** statement has the form

case *selectorExpression* of

caseList$_1$: *statement*$_1$;

⋮

caseList$_n$: *statement*$_n$;

end

where *selectorExpression* is any expression of an ordinal type (string types are invalid) and each *caseList* is one of the following:

- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus `7`, `True`, `4 + 5 * 3`, `'A'`, and `Integer('A')` can all be used as *caseList*s, but variables and most function calls cannot. (A few built-in functions like *Hi* and *Lo* can occur in a *caseList*. See “Constant expressions” on page 5-41.)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item*$_1$, ..., *item*$_n$, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the **case** statement; subranges and lists cannot overlap. A **case** statement can have a final **else** clause:

```
case selectorExpression of
  caseList1: statement1;
  ⋮
  caseListn: statementn;
else
  statements;
end
```

where *statements* is a semicolon-delimited sequence of statements. When a **case** statement is executed, at most one of *statement*$_1$ ... *statement*$_n$ is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the *statement* to be used. If none of the *caseList*s has the same value as *selectorExpression*, then the statements in the **else** clause (if there is one) are executed.

The **case** statement

```
case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end;
```

is equivalent to the nested conditional

```
if I in [1..5] then
  Caption := 'Low'
  else if I in [6..10] then
    Caption := 'High'
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Out of range'
      else
        Caption := '';
```

Other examples of **case** statements:

```
case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X := 3;
  Yellow, Orange, Black: X := 0;
```

```
end;

case Selection of
  Done: Form1.Close;
  Compute: CalculateTotal(UnitCost, Quantity);
else
  Beep;
end;
```

Control loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Object Pascal has three kinds of control loop: **repeat** statements, **while** statements, and **for** statements.

You can use the standard *Break* and *Continue* procedures to control the flow of a **repeat**, **while**, or **for** statement. *Break* terminates the statement in which it occurs, while *Continue* begins executing the next iteration of the sequence. For more information about these procedures, see the online Help.

Repeat statements

The syntax of a **repeat** statement is

repeat *statement*$_1$; ...; *statement*$_n$; until *expression*

where *expression* returns a Boolean value. (The last semicolon before **until** is optional.) The **repeat** statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns *True*, the **repeat** statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of **repeat** statements include

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While statements

A **while** statement is similar to a **repeat** statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a **while** statement is

while *expression* do *statement*

where *expression* returns a Boolean value and *statement* can be a compound statement. The **while** statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns *True*, execution continues.

Examples of **while** statements include

```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

For statements

A **for** statement, unlike a **repeat** or **while** statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a **for** statement is

`for` *counter* `:=` *initialValue* `to` *finalValue* `do` *statement*

or

`for` *counter* `:=` *initialValue* `downto` *finalValue* `do` *statement*

where

- *counter* is a local variable (declared in the block containing the **for** statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with *counter*.
- *statement* is a simple or structured statement that does not change the value of *counter*.

The **for** statement assigns the value of *initialValue* to *counter*, then executes *statement* repeatedly, incrementing or decrementing *counter* after each iteration. (The **for...to** syntax increments *counter*, while the **for...downto** syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the **for** statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a **for...to** statement, or less than *finalValue* in a **for...downto** statement, then *statement* is never executed. After the **for** statement terminates, the value of *counter* is undefined.

For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the **for...to** statement is almost, but not quite, equivalent to this **while** construction:

```
begin
  counter := initialValue;
  while counter <= finalValue do
  begin
    statement;
    counter := Succ(counter);
  end;
end
```

The difference between this construction and the **for...to** statement is that the **while** loop re-evaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of **for** statements:

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];

for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);

for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Check(C);
```

Blocks and scope

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

Blocks

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```
declarations
begin
  statements
end
```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program block, the *declarations* section can also include one or more **exports** clauses (see Chapter 9, "Libraries and packages").

For example, in a function declaration like

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  ⋮
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. *Ch*, *L*, *Source*, and *Dest* are local variables; their declarations apply only to the *UpperCase* function block and override—in this block only—any declarations of the same identifiers that may occur in the program block or in the interface or implementation section of a unit.

Scope

An identifier, such as a variable or function name, can be used only within the *scope* of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope—especially identifiers declared in functions and procedures—are sometimes called *local*, while identifiers with wider scope are called *global*.

The rules that determine identifier scope are summarized below.

If the identifier is declared in ...	its scope extends ...
the declaration of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Chapter 3, "Programs and units".)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See "Records" on page 5-21.)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Chapter 7, "Classes and objects".)

Naming conflicts

When one block encloses another, the former is called the *outer block* and the latter the *inner block*. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration overrides the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called *MaxValue* in the interface section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of *MaxValue* in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a **uses** clause imposes a new scope that encloses the remaining units used and the program or unit containing the **uses** clause. The first unit in a **uses** clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their interface sections, an unqualified reference to the identifier selects the declaration in the innermost scope—that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the **uses** clause that does declare the identifier.

The *System* unit is used automatically by every program or unit. The declarations in *System*, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and by-pass an inner declaration by using a qualified identifier (see "Qualified identifiers" on page 4-2) or a **with** statement (see "With statements" on page 4-20).

Chapter

5

Data types, variables, and constants

A *type* is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

Object Pascal is a “strongly typed” language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include *typecasting* (see “Typecasts” on page 4-14), *pointers* (see “Pointers and pointer types” on page 5-25), *variants* (see “Variant types” on page 5-30), *variant parts* in records (see “Variant parts in records” on page 5-22), and *absolute addressing* of variables (see “Absolute addresses” on page 5-38).

About types

There are several ways to categorize Object Pascal data types:

- Some types are *predefined* (or *built-in*); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either *fundamental* or *generic*. The range and format of a fundamental type is the same in all implementations of Object Pascal, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It’s a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage

format from one implementation of a generic type to the next could cause compatibility problems—for example, if you are streaming data to a file.

- Types can be classified as *simple*, *string*, *structured*, *pointer*, *procedural*, or *variant*. In addition, type identifiers themselves can be regarded as belonging to a special "type" because they can be passed as parameters to certain functions (such as *High*, *Low*, and *SizeOf*).

The outline below shows the taxonomy of Object Pascal data types.

- **simple**
  - ordinal
    - integer
    - character
    - Boolean
    - enumerated
    - subrange
  - real
- **string**
- **structured**
  - set
  - array
  - record
  - file
  - class
  - class reference
  - interface
- **pointer**
- **procedural**
- **variant**
- **type identifier**

The standard function *SizeOf* operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) used to store data of the specified type. For example, `SizeOf(Longint)` returns 4, since a *Longint* variable uses four bytes of memory.

Type declarations are illustrated in the sections that follow. For general information about type declarations, see "Declaring types" on page 5-36.

Simple types

Simple types, which include *ordinal* types and *real* types, define ordered sets of values.

Ordinal types

Ordinal types include *integer*, *character*, *Boolean*, *enumerated*, and *subrange* types. An ordinal type defines an ordered set of values in which each value except the first has a unique *predecessor* and each value except the last has a unique *successor*. Further, each value has an *ordinality* which determines the ordering of the type. In most cases, if a value has ordinality *n*, its predecessor has ordinality *n*–1 and its successor has ordinality *n*+1.

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
Ord	ordinal expression	ordinality of expression's value	Does not take *Int64* arguments.
Pred	ordinal expression	predecessor of expression's value	Do not use on properties that have a **write** procedure.
Succ	ordinal expression	successor of expression's value	Do not use on properties that have a **write** procedure.
High	ordinal type identifier or variable of ordinal type	highest value in type	Also operates on short-string types and arrays.
Low	ordinal type identifier or variable of ordinal type	lowest value in type	Also operates on short-string types and arrays.

For example, `High(Byte)` returns 255 because the highest value of type *Byte* is 255, and `Succ(2)` returns 3 because 3 is the successor of 2.

The standard procedures *Inc* and *Dec* increment and decrement the value of an ordinal variable. For example, `Inc(I)` is equivalent to `I := Succ(I)` and, if *I* is an integer variable, to `I := I + 1`.

Integer types

An integer type represents a subset of the whole numbers. The generic integer types are *Integer* and *Cardinal*; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the current 32-bit Object Pascal compiler.

Table 5.1 Generic integer types for 32-bit implementations of Object Pascal

Type	Range	Format
Integer	–2147483648..2147483647	signed 32-bit
Cardinal	0..4294967295	unsigned 32-bit

Fundamental integer types include *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word*, and *Longword*.

Table 5.2 Fundamental integer types

Type	Range	Format
Shortint	–128..127	signed 8-bit
Smallint	–32768..32767	signed 16-bit
Longint	–2147483648..2147483647	signed 32-bit
Int64	$-2^{63}..2^{63}-1$	signed 64-bit
Byte	0..255	unsigned 8-bit
Word	0..65535	unsigned 16-bit
Longword	0..4294967295	unsigned 32-bit

In general, arithmetic operations on integers return a value of type *Integer*—which, in its current implementation, is equivalent to the 32-bit *Longint*. Operations return a value of type *Int64* only when performed on an *Int64* operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
  ⋮
I := High(Integer);
J := I + 1;
```

To get an *Int64* return value in this situation, cast *I* as *Int64*:

```
 ⋮
J := Int64(I) + 1;
```

For more information, see “Arithmetic operators” on page 4-6.

Note Most standard routines that take integer arguments truncate *Int64* values to 32 bits. However, the *High*, *Low*, *Succ*, *Pred*, *Inc*, *Dec*, *IntToStr*, and *IntToHex* routines fully support *Int64* arguments. Also, the *Round*, *Trunc*, *StrToInt64*, and *StrToInt64Def* functions return *Int64* values. A few routines—including *Ord*—cannot take *Int64* values at all.

When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the *Shortint* type has the range –128..127; hence, after execution of the code

```
var I: Shortint;
 ⋮
I := High(Shortint);
I := I + 1;
```

the value of *I* is –128. If compiler range-checking is enabled, however, this code generates a runtime error.

Character types

The fundamental character types are *AnsiChar* and *WideChar*. *AnsiChar* values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. *AnsiChar* was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

WideChar characters use more than one byte to represent every character. In the current implementations, *WideChar* is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

The generic character type is *Char*, which is equivalent to *AnsiChar*. Because the implementation of *Char* is subject to change, it's a good idea to use the standard function *SizeOf* rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

A string constant of length 1, such as `'A'`, can denote a character value. The predefined function *Chr* returns the character value for any integer in the range of *AnsiChar* or *WideChar*; for example, `Chr(65)` returns the letter *A*.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

```
var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
end;
```

Letter has the value *A* (ASCII 65).

For more information about Unicode characters, see "About extended character sets" on page 5-13 and "Working with null-terminated strings" on page 5-13.

Boolean types

The four predefined Boolean types are *Boolean*, *ByteBool*, *WordBool*, and *LongBool*. *Boolean* is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A *Boolean* variable occupies one byte of memory, a *ByteBool* variable also occupies one byte, a *WordBool* variable occupies two bytes (one word), and a *LongBool* variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants *True* and *False*. The following relationships hold.

Boolean	**ByteBool, WordBool, LongBool**
False < *True*	*False* <> *True*
Ord(*False*) = 0	*Ord*(*False*) = 0
Ord(*True*) = 1	*Ord*(*True*) <> 0
Succ(*False*) = *True*	*Succ*(*False*) = *True*
Pred(*True*) = *False*	*Pred*(*False*) = *True*

A value of type *ByteBool*, *LongBool*, or *WordBool* is considered *True* when its ordinality is nonzero. If such a value appears in a context where a *Boolean* is expected, the compiler automatically converts any value of nonzero ordinality to *True*.

The remarks above refer to the ordinality of Boolean values—not to the values themselves. In Object Pascal, Boolean expressions cannot be equated with integers or reals. Hence, if *X* is an integer variable, the statement

```
if X then ...;
```

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```
if X <> 0 then ...;       { use longer expression that returns Boolean value }

var OK: Boolean           { use Boolean variable }
 ⋮
if X <> 0 then OK := True;
if OK then ...;
```

Enumerated types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

type *typeName* = (*val*$_1$, ..., *val*$_n$)

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called *Suit* whose possible values are *Club*, *Diamond*, *Heart*, and *Spade*, where `Ord(Club)` returns 0, `Ord(Diamond)` returns 1, and so forth.

When you declare an enumerated type, you are declaring each *val* to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock);
```

Unfortunately, *Click* is also the name of a method defined for *TControl* and all of the objects in the VCL and/or CLX that descend from it. So if you're writing an application and you create an event handler like

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
  ⋮
  Thing := Click;
  ⋮
end;
```

you'll get a compilation error; the compiler interprets *Click* within the scope of the procedure as a reference to *TForm*'s *Click* method. You can work around this by qualifying the identifier; thus, if *TSound* is declared in *MyUnit*, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe);
```

You can use the (*val*$_1$, ..., *val*$_n$) construction directly in variable declarations, as if it were a type name:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare *MyCard* this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

generates a compilation error. But

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```
type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;
```

Enumerated types with explicitly assigned ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with = *constantExpression*, where *constantExpression* is a constant expression that evaluates to an integer. (See "Constant expressions" on page 5-41) For example,

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

defines a type called *Size* whose possible values include *Small*, *Medium*, and *Large*, where `Ord(Small)` returns 5, `Ord(Medium)` returns 10, and `Ord(Large)` returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration.

In the example above, the *Size* type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.) Only three of these values have names, but the others are accessible through typecasts and through routines such as *Pred*, *Succ*, *Inc*, and *Dec*. In the following example, “anonymous” values in the range of *Size* are assigned to the variable *X*.

```
var X: Size;
X := Small;    // Ord(X) = 5
X := Size(6);  // Ord(X) = 6
Inc(X);        // Ord(X) = 7
```

Any value that isn’t explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn’t assigned an ordinality, its ordinality is 0. Hence, given the declaration

```
type SomeEnum = (e1, e2, e3 = 1);
```

SomeEnum has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because *e2* and *e3* have the same ordinality, they represent the same value.

Subrange types

A subrange type represents a subset of the values in another ordinal type (called the *base type*). Any construction of the form *Low..High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here *TMyColors* includes the values *Green*, *Yellow*, *Orange*, *Purple*, and *White*.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *Low..High* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example above, if *Color* is a variable that holds the value *Green*, `Ord(Color)` returns 2 regardless of whether *Color* is of type *TColors* or *TMyColors*.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type Percentile = 0..99;
var I: Percentile;
⋮
I := 100;
```

produces an error,

```
⋮
I := 99;
Inc(I);
```

assigns the value 100 to *I* (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after **=** is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;
type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real types

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types.

Table 5.3 Fundamental real types

Type	Range	Significant digits	Size in bytes
Real48	2.9 x 10^{-39} .. 1.7 x 10^{38}	11–12	6
Single	1.5 x 10^{-45} .. 3.4 x 10^{38}	7–8	4
Double	5.0 x 10^{-324} .. 1.7 x 10^{308}	15–16	8
Extended	3.6 x 10^{-4951} .. 1.1 x 10^{4932}	19–20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19–20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19–20	8

The generic type *Real*, in its current implementation, is equivalent to *Double*.

Table 5.4 Generic real types

Type	Range	Significant digits	Size in bytes
Real	5.0×10^{-324} .. 1.7×10^{308}	15–16	8

Note The six-byte *Real48* type was called *Real* in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte *Real* type, you may want to change it to *Real48*. You can also use the **{$REALCOMPATIBILITY ON}** compiler directive to turn *Real* back into the six-byte type.

The following remarks apply to fundamental real types.

- *Real48* is maintained for backward compatibility. Since its storage format is not native to the Intel CPU family, it results in slower performance than other floating-point types.
- *Extended* offers greater precision than other real types but is less portable. Be careful using *Extended* if you are creating data files to share across platforms.
- The *Comp* (computational) type is native to the Intel CPU and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a *Comp* value.) *Comp* is maintained for backward compatibility only. Use the *Int64* type for better performance.
- *Currency* is a fixed-point data type that minimizes rounding errors in monetary calculations. It is stored as a scaled 64-bit integer with the four least-significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, *Currency* values are automatically divided or multiplied by 10000.

String types

A string represents a sequence of characters. Object Pascal supports the following predefined string types.

Table 5.5 String types

Type	Maximum length	Memory required	Used for
ShortString	255 characters	2 to 256 bytes	backward compatibility
AnsiString	~2^{31} characters	4 bytes to 2GB	8-bit (ANSI) characters
WideString	~2^{30} characters	4 bytes to 2GB	Unicode characters; multiuser servers and multi-language applications

AnsiString, sometimes called the *long string*, is the preferred type for most purposes.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a function or procedure (as **var** and **out** parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type (see “Typecasts” on page 4-14).

The reserved word **string** functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable *S* that holds a string. In the default **{$H+}** state, the compiler interprets **string** (when it appears without a bracketed number after it) as *AnsiString*. Use the **{$H–}** directive to turn **string** into *ShortString*.

The standard function *Length* returns the number of characters in a string. The *SetLength* procedure adjusts the length of a string. See the online Help for details.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value. For example, “AB” is greater than “A”; that is, `'AB' > 'A'` returns *True*. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If *S* is a string variable and *i* an integer expression, `S[i]` represents the *i*th character—or, strictly speaking, the *i*th byte—in *S*. For a *ShortString* or *AnsiString*, `S[i]` is of type *AnsiChar*; for a *WideString*, `S[i]` is of type *WideChar*. The statement `MyString[2] := 'A';` assigns the value *A* to the second character of *MyString*. The following code uses the standard *UpCase* function to convert *MyString* to uppercase.

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
  begin
    MyString[I] := UpCase(MyString[I]);
    I := I - 1;
  end;
end;
```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as **var** parameters, because this results in inefficient code.

You can assign the value of a string constant—or any other expression that returns a string—to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';             { space }
MyString := '';              { empty string }
```

For more information, see “Character strings” on page 4-4 and “String operators” on page 4-9.

Short strings

A *ShortString* is 0 to 255 characters long. While the length of a *ShortString* can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If *S* is a *ShortString* variable, `Ord(S[0])`, like `Length(S)`, returns the length of *S*; assigning a value to `S[0]`, like calling *SetLength*, changes the length of *S*. *ShortString* uses 8-bit ANSI characters and is maintained for backward compatibility only.

Object Pascal supports short-string types—in effect, subtypes of *ShortString*—whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word **string**. For example,

```
var MyString: string[100];
```

creates a variable called *MyString* whose maximum length is 100 characters. This is equivalent to the declarations

```
type CString = string[100];
var MyString: CString;
```

Variables declared in this way allocate only as much memory as the type requires—that is, the specified maximum length plus one byte. In our example, *MyString* uses 101 bytes, as compared to 256 bytes for a variable of the predefined *ShortString* type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions *High* and *Low* operate on short-string type identifiers and variables. *High* returns the maximum length of the short-string type, while *Low* returns zero.

Long strings

AnsiString, also called a *long string*, represents a dynamically allocated string whose maximum length is limited only by available memory. It uses 8-bit ANSI characters.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty—that is, when it contains a zero-length string—the pointer is **nil** and the string uses no additional storage. When the variable is nonempty, it points to a dynamically allocated block of memory that contains the string value, a 32-bit length indicator, and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called *reference-counting*. When indexing is used to change the value of a single character in a string, a copy of the string is made if—but only if—its reference count is greater than one. This is called *copy-on-write* semantics.

WideString

The *WideString* type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to *AnsiString*.

On Win32, *WideString* is compatible with the COM *BSTR* type. Borland development tools have support features that convert *AnsiString* values to *WideString*, but you may need to explicitly cast or convert your strings to *WideString*.

About extended character sets

Windows and Linux both support *single-byte* and *multibyte* character sets as well as *Unicode*. With a single-byte character set (SBCS), each byte in a string represents one character. The ANSI character set used by many Western operating systems is a single-byte character set.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the *lead byte*. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages, while the UTF-8 character set used by Linux is a multibyte encoding of Unicode.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called *wide characters* and *wide character strings*. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi/Kylix supports UCS-2 on both platforms.

Object Pascal supports single-byte and multibyte characters and strings through the *Char*, *PChar*, *AnsiChar*, *PAnsiChar*, and *AnsiString* types. Indexing of multibyte strings is not reliable, since `S[i]` represents the *i*th byte (not necessarily the *i*th character) in *S*. However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with *Ansi-*. For example, the multibyte version of *StrPos* is *AnsiStrPos*.) Multibyte character support is operating-system dependent and based on the current locale.

Object Pascal supports Unicode characters and strings through the *WideChar*, *PWideChar*, and *WideString* types.

Working with null-terminated strings

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on *null-terminated* strings. A null-terminated string is a zero-based array of characters that ends with NULL (#0); since the array has no length indicator, the first NULL character marks the end of the string. You can use Object Pascal constructions and special routines in the *SysUtils* unit (see Chapter 8, "Standard routines and I/O") to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

With extended syntax enabled (**{$X+}**), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won't work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to #0. For more information about arrays, see "Arrays" on page 5-18.

Using pointers, arrays, and string constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See "Pointers and pointer types" on page 5-25.) String constants are assignment-compatible with the *PChar* and *PWideChar* types, which represent pointers to null-terminated arrays of *Char* and *WideChar* values. For example,

```
var P: PChar;
 ⋮
P := 'Hello world!';
```

points *P* to an area of memory that contains a null-terminated copy of "Hello world!" This is equivalent to

```
const TempString: array[0..12] of Char = 'Hello world!'#0;
var P: PChar;
 ⋮
P := @TempString;
```

You can also pass string constants to any function that takes value or **const** parameters of type *PChar* or *PWideChar*—for example `StrUpper('Hello world!')`. As with assignments to a *PChar*, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize *PChar* or *PWideChar* constants with string literals, alone or in a structured type. Examples:

```
const
Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Zero-based character arrays are compatible with *PChar* and *PWideChar*. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

This code calls *SomeProcedure* twice with the same value.

A character pointer can be indexed as if it were an array. In the example above, `MyPointer[0]` returns *H*. The index specifies an offset added to the pointer before it is dereferenced. (For *PWideChar* variables, the index is automatically multiplied by two.) Thus, if *P* is a character pointer, *P*`[0]` is equivalent to *P*^ and specifies the first character in the array, *P*`[1]` specifies the second character in the array, and so forth; *P*`[-1]` specifies the "character" immediately to the left of *P*`[0]`. *The compiler performs no range checking on these indexes.*

The *StrUpper* function illustrates the use of pointer indexing to iterate through a null-terminated string:

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

Mixing Pascal strings and null-terminated strings

You can mix long strings (*AnsiString* values) and null-terminated strings (*PChar* values) in expressions and assignments, and you can pass *PChar* values to functions or procedures that take long-string parameters. The assignment *S* `:=` *P*, where *S* is a string variable and *P* is a *PChar* expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a *PChar*, the *PChar* operand is converted to a long string.

You can cast a *PChar* value as a long string. This is useful when you want to perform a string operation on two *PChar* values. For example,

```
S := string(P1) + string(P2);
```

You can also cast a long string as a null-terminated string. The following rules apply.

- If *S* is a long-string expression, `PChar(S)` casts *S* as a null-terminated string; it returns a pointer to the first character in *S*.

  On Windows: For example, if *Str1* and *Str2* are long strings, you could call the Win32 API *MessageBox* function like this:

  ```
  MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
  ```

  (The declaration of *MessageBox* is in the *Windows* interface unit.)

  On Linux: For example, if *Str* is a long string, you could call the *opendir* system function like this:

  ```
  opendir(PChar(Str));
  ```

  (The declaration of *opendir* is in the *Libc* interface unit.)

- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if *S* is empty, the typecast returns **nil**.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the long string only when all of the following conditions are satisfied.
  - The expression cast is a long-string *variable*.
  - The string is not empty.
  - The string is unique—that is, has a reference count of one. To guarantee that the string is unique, call the *SetLength*, *SetString*, or *UniqueString* procedure.
  - The string has not been modified since the typecast was made.
  - The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing *WideString* values with *PWideChar* values.

Structured types

Instances of a structured type hold more than one value. Structured types include *sets*, *arrays*, *records*, and *files* as well as *class*, *class-reference*, and *interface* types. (For information about class and class-reference types, see Chapter 7, “Classes and objects”. For information about interfaces, see Chapter 10, “Object interfaces”.) Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word **packed** to implement compressed data storage. For example,

```
type TNumbers = packed array[1..100] of Real;
```

Using **packed** slows data access and, in the case of a character array, affects type compatibility. For more information, see Chapter 11, “Memory management”.

Sets

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the *base type*; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

```
set of baseType
```

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

create a set type called *TIntSet* whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

```
type TIntSet = set of 1..250;
```

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
  ⋮
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

You can also use the **set of ...** construction directly in variable declarations:

```
var MySet: set of 'a'..'z';
  ⋮
MySet := ['a','b','c'];
```

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The **in** operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by `[]`. For more information about sets, see “Set constructors” on page 4-13 and “Set operators” on page 4-10.

Arrays

An array represents an indexed collection of elements of the same type (called the *base type*). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

Static arrays

Static array types are denoted by constructions of the form

```
array[indexType1, ..., indexTypen] of baseType
```

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexType*s index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexType*s. In practice, *indexType*s are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

```
var MyArray: array[1..100] of Char;
```

declares a variable called *MyArray* that holds an array of 100 character values. Given this declaration, `MyArray[3]` denotes the third character in *MyArray*. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type TMatrix = array[1..10, 1..50] of Real;
```

Whichever way *TMatrix* is declared, it represents an array of 500 real values. A variable *MyMatrix* of type *TMatrix* can be indexed like this: `MyMatrix[2,45]`; or like this: `MyMatrix[2][45]`. Similarly,

```
packed array[Boolean,1..10,TShoeSize] of Integer;
```

is equivalent to

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

The standard functions *Low* and *High* operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function *Length* returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of *Char* values is called a *packed string*. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See "Type compatibility and identity" on page 5-34.

An array type of the form `array[0..x] of Char` is called a *zero-based character array*. Zero-based character arrays are used to store null-terminated strings and are compatible with *PChar* values. See "Working with null-terminated strings" on page 5-13.

Dynamic arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the *SetLength* procedure. Dynamic-array types are denoted by constructions of the form

```
array of baseType
```

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for *MyFlexibleArray*. To create the array in memory, call *SetLength*. For example, given the declaration above,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign **nil** to a variable that references the array or pass the variable to *Finalize*; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays of length 0 have the value **nil**. Do not apply the dereference operator (**^**) to a dynamic-array variable or pass it to the *New* or *Dispose* procedure.

If *X* and *Y* are variables of the same dynamic-array type, *X* `:=` *Y* points *X* to the same array as *Y*. (There is no need to allocate memory for *X* before performing this operation.) Unlike strings and static arrays, dynamic arrays are not automatically copied before they are written to. For example, after this code executes—

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

—the value of `A[0]` is 2. (If *A* and *B* were static arrays, `A[0]` would still be 1.)

Assigning to a dynamic-array index (for example, `MyFlexibleArray[2] := 7`) does not reallocate the array. Out-of-range indexes are not reported at compile time.

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

`A = B` returns *False* but `A[0] = B[0]` returns *True*.

To truncate a dynamic array, pass it to *SetLength* or *Copy* and assign the result back to the array variable. (The *SetLength* procedure is usually faster.) For example, if *A* is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of *A*.

Once a dynamic array has been allocated, you can pass it to the standard functions *Length*, *High*, and *Low*. *Length* returns the number of elements in the array, *High* returns the array's highest index (that is, *Length*–1), and *Low* returns 0. In the case of a zero-length array, *High* returns –1 (with the anomalous consequence that *High* < *Low*).

Note In some function and procedure declarations, array parameters are represented as `array of` *baseType*, without any index types specified. For example,

```
function CheckStrings(A: array of string): Boolean;
```

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically. See "Open array parameters" on page 6-15.

Multidimensional dynamic arrays

To declare multidimensional dynamic arrays, use iterated **array of ...** constructions. For example,

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

declares a two-dimensional array of strings. To instantiate this array, call *SetLength* with two integer arguments. For example, if *I* and *J* are integer-valued variables,

```
SetLength(Msgs,I,J);
```

allocates an *I*-by-*J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call *SetLength*, passing it parameters for the first *n* dimensions of the array. For example,

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

allocates ten rows for *Ints* but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

```
SetLength(Ints[2], 5);
```

makes the third column of *Ints* five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column—for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the *IntToStr* function declared in the *SysUtils* unit) to create a triangular matrix of strings.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
```

```
    begin
      SetLength(A[I], I);
      for J := Low(A[I]) to High(A[I]) do
        A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
    end;
  end;
```

Array types and assignments

Arrays are assignment-compatible only if they are of the same type. Because Pascal uses name-equivalence for types, the following code will not compile.

```
var
  Int1: array[1..10] of Integer;
  Int2: array[1..10] of Integer;
  ⋮
Int1 := Int2;
```

To make the assignment work, declare the variables as

```
var Int1, Int2: array[1..10] of Integer;
```

or

```
type IntArray = array[1..10] of Integer;
var
  Int1: IntArray;
  Int2: IntArray;
```

Records

A record (analogous to a *structure* in some languages) represents a heterogeneous set of elements. Each element is called a *field*; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
  fieldList1: type1;
  ⋮
  fieldListn: typen;
end
```

where *recordTypeName* is a valid identifier, each *type* denotes a type, and each *fieldList* is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called *TDateRec*.

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

Each *TDateRec* contains three fields: an integer value called *Year*, a value of an enumerated type called *Month*, and another integer between 1 and 31 called *Day*. The identifiers *Year*, *Month*, and *Day* are the *field designators* for *TDateRec*, and they behave like variables. The *TDateRec* type declaration, however, does not allocate any memory for the *Year*, *Month*, and *Day* fields; memory is allocated when you instantiate the record, like this:

```
var Record1, Record2: TDateRec;
```

This variable declaration creates two instances of *TDateRec*, called *Record1* and *Record2*.

You can access the fields of a record by qualifying the field designators with the record's name:

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

Or use a **with** statement:

```
with Record1 do
begin
  Year := 1904;
  Month := Jun;
  Day := 16;
end;
```

You can now copy the values of *Record1*'s fields to *Record2*:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the **record ...** construction directly in variable declarations:

```
var S: record
  Name: string;
  Age: Integer;
end;
```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

Variant parts in records

A record type can have a *variant* part, which looks like a **case** statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```
type recordTypeName = record
  fieldList1: type1;
  ⋮
  fieldListn: typen;
case tag: ordinalType of
  constantList1: (variant1);
  ⋮
  constantListn: (variantn);
end;
```

The first part of the declaration—up to the reserved word **case**—is the same as that of a standard record type. The remainder of the declaration—from **case** to the optional final semicolon—is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (**:**) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type o*rdinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantList*s.
- Each *variant* is a comma-delimited list of declarations resembling the *fieldList*: *type* constructions in the main part of the record type. That is, a *variant* has the form

  ```
  fieldList1: type1;
    ⋮
  fieldListn: typen;
  ```

  where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each *type* denotes a type, and the final semicolon is optional. The *types* must not be long strings, dynamic arrays, variants (that is, *Variant* types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several *variant*s which share the same space in memory. You can read or write to any field of any *variant* at any time; but if you write to a field in one *variant* and then to a field in another *variant*, you may be overwriting your own data. The *tag*, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```
type
  TEmployee = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Salaried: Boolean of
    True: (AnnualSalary: Currency);
    False: (HourlyWage: Currency);
end;
```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of *TEmployee*, there is no reason to allocate enough memory for both fields. In this case, the only difference between the *variant*s is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```
type
  TPerson = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Citizen: Boolean of
    True: (Birthplace: string[40]);
    False: (Country: string[20];
            EntryPort: string[20];
            EntryDate, ExitDate: TDate);
  end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
      Circle: (Radius: Real);
      Ellipse, Other: ();
  end;
```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest *variant*. The optional *tag* and the *constantList*s (like *Rectangle*, *Triangle*, and so forth in the last example above) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit *Real* as the first field in one *variant* and a 32-bit *Integer* as the first field in another, you can assign a value to the *Real* field and then read back the first 32 bits of it as the value of the *Integer* field (passing it, say, to a function that requires integer parameters).

File types

A file is an ordered set of elements of the same type. Standard I/O routines use the predefined *TextFile* or *Text* type, which represents a file containing characters organized into lines. For more information about file input and output, see Chapter 8, "Standard routines and I/O".

To declare a file type, use the syntax

```
type fileTypeName = file of type
```

where *fileTypeName* is any valid identifier and *type* is a fixed-size type. Pointer types—whether implicit or explicit—are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```
type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;
  PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the **file of ...** construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word **file** by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see "Untyped files" on page 8-4.

Files are not allowed in arrays or records.

Pointers and pointer types

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it *points* to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure.

Pointers are *typed* to indicate the kind of data stored at the addresses they hold. The general-purpose *Pointer* type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

Overview of pointers

To see how pointers work, look at the following example.

```
1    var
2      X, Y: Integer;   // X and Y are Integer variables
3      P: ^Integer;     // P points to an Integer
4    begin
5      X := 17;         // assign a value to X
6      P := @X;         // assign the address of X to P
7      Y := P^;         // dereference P; assign the result to Y
8    end;
```

Line 2 declares *X* and *Y* as variables of type *Integer*. Line 3 declares *P* as a pointer to an *Integer* value; this means that *P* can point to the location of *X* or *Y*. Line 5 assigns a value to *X*, and line 6 assigns the address of *X* (denoted by `@X`) to *P*. Finally, line 7 retrieves the value at the location pointed to by *P* (denoted by `^P`) and assigns it to *Y*. After this code executes, *X* and *Y* have the same value, namely 17.

The @ operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see "The @ operator" on page 4-12 and "Procedural types in statements and expressions" on page 5-29.

The symbol **^** has two purposes, both of which are illustrated in our example. When it appears before a type identifier—

^typeName

—it denotes a type that represents pointers to variables of type *typeName*. When it appears after a pointer variable—

*pointer^*

—it *dereferences* the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another—something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand Object Pascal, since pointers often operate behind the scenes in code where they don't appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Object Pascal's strict data typing. By referencing a variable with an all-purpose *Pointer*, casting the *Pointer* to a more specific type, and then dereferencing it, you can treat the data stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ⋮
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from *R* to *I*, without converting it.

In addition to assigning the result of an @ operation, you can use several standard routines to give a value to a pointer. The *New* and *GetMem* procedures assign a memory address to an existing pointer, while the *Addr* and *Ptr* functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `P1^.Data^`.

The reserved word **nil** is a special constant that can be assigned to any pointer. When **nil** is assigned to a pointer, the pointer doesn't reference anything.

Pointer types

You can declare a pointer to any type, using the syntax

`type` *pointerTypeName* `= ^`*type*

When you define a record or other data type, it's a common practice also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Standard pointer types exist for many purposes. The most versatile is *Pointer*, which can point to data of any kind. But a *Pointer* variable cannot be dereferenced; placing the **^** symbol after a *Pointer* variable causes a compilation error. To access the data referenced by a *Pointer* variable, first cast it to another pointer type and then dereference it.

Character pointers

The fundamental types *PAnsiChar* and *PWideChar* represent pointers to *AnsiChar* and *WideChar* values, respectively. The generic *PChar* represents a pointer to a *Char* (that is, in its current implementation, to an *AnsiChar*). These character pointers are used to manipulate null-terminated strings. (See "Working with null-terminated strings" on page 5-13.)

Other standard pointer types

The *System* and *SysUtils* units declare many standard pointer types that are commonly used.

Table 5.6 Selected pointer types declared in System and SysUtils

Pointer type	Points to variables of type
PAnsiString, *PString*	*AnsiString*
PByteArray	*TByteArray* (declared in *SysUtils*). Used to typecast dynamically allocated memory for array access.
PCurrency, *PDouble*, *PExtended*, *PSingle*	*Currency*, *Double*, *Extended*, *Single*
PInteger	*Integer*
POleVariant	*OleVariant*
PShortString	*ShortString*. Useful when porting legacy code that uses the old *PString* type.
PTextBuf	*TTextBuf* (declared in *SysUtils*). *TTextBuf* is the internal buffer type in a *TTextRec* file record.)
PVarRec	*TVarRec* (declared in *System*)
PVariant	*Variant*

Table 5.6 Selected pointer types declared in System and SysUtils (continued)

Pointer type	Points to variables of type
PWideString	*WideString*
PWordArray	*TWordArray* (declared in *SysUtils*). Used to typecast dynamically allocated memory for arrays of 2-byte values.

Procedural types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called *Calc* that takes two integer parameters and returns an integer:

```
function Calc(X,Y: Integer): Integer;
```

You can assign the *Calc* function to the variable *F*:

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

If you take any procedure or function heading and remove the identifier after the word **procedure** or **function**, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the example above) or to declare new types:

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var
  F: TIntegerFunction;            { F is a parameterless function that returns an integer }
  Proc: TProcedure;               { Proc is a parameterless procedure }
  SP: TStrProc;                   { SP is a procedure that takes a string parameter }
  M: TMathFunc;                   { M is a function that takes a Double (real) parameter
                                      and returns a Double }
procedure FuncProc(P: TIntegerFunction);  { FuncProc is a procedure whose only parameter
                                              is a parameterless integer-valued function }
```

The variables above are all *procedure pointers*—that is, pointers to the address of a procedure or function. If you want to reference a method of an instance object (see Chapter 7, "Classes and objects"), you need to add the words **of object** to the procedural type name. For example

```
type
  TMethod = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

These types represent *method pointers*. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ⋮
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent
```

we could make the following assignment.

```
OnClick := MainForm.ButtonClick;
```

Two procedural types are compatible if they have

- the same calling convention,
- the same return value (or no return value), and
- the same number of parameters, with identically typed parameters in corresponding positions. (Parameter names do not matter.)

Procedure pointer types are always incompatible with method pointer types. The value **nil** can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like *Length* as a procedural value, write a wrapper for it:

```
function FLength(S: string): Integer;
begin
  Result := Length(S);
end;
```

Procedural types in statements and expressions

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```
var
  F: function(X: Integer): Integer;
  I: Integer;
function SomeFunction(X: Integer): Integer;
 ⋮
F := SomeFunction;  // assign SomeFunction to F
I := F(4);          // call function; assign result to I
```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```
var
  F, G: function: Integer;
  I: Integer;
```

```
function SomeFunction: Integer;
 ⋮
F := SomeFunction;  // assign SomeFunction to F
G := F;             // copy F to G
I := G;             // call function; assign result to I
```

The first statement assigns a procedural value to *F*. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to *I*. Because *I* is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of *F* results in a function call; the compiler calls the function pointed to by *F*, then calls the function *MyFunction*, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where *F* references a procedure (which doesn't return a value), or where *F* references a function that requires parameters, the statement above causes a compilation error. To compare the procedural value of *F* with *MyFunction*, use

```
if @F = @MyFunction then ...;
```

`@F` converts *F* into an untyped pointer variable that contains an address, and `@MyFunction` returns the address of *MyFunction*.

To get the memory address of a procedural variable (rather than the address stored in it), use @@. For example, `@@F` returns the address of *F*.

The @ operator can also be used to assign an untyped pointer value to a procedural variable. For example,

```
var StrComp: function(Str1, Str2: PChar): Integer;
 ⋮
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

calls the *GetProcAddress* function and points *StrComp* to the result.

Any procedural variable can hold the value **nil**, which means that it points to nothing. But attempting to *call* a **nil**-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function *Assigned*:

```
if Assigned(OnClick) then OnClick(X);
```

Variant types

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type *Variant*, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See Chapter 10, "Object interfaces".) They can hold dynamic arrays, and they can hold a special kind of static array called a *variant array*. (See "Variant arrays" on page 5-33.) Variants can mix with other variants and with integer, real, string, and Boolean values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if *V* is a variant that holds a string value, the construction `V[1]` causes a runtime error.

You can define custom Variants that extend the Variant type to hold arbitrary values. For example, you can define a Variant string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the *TCustomVariantType* class.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value *Unassigned*. The special value *Null* indicates unknown or missing data.

The standard function *VarType* returns a variant's type code. The *varTypeMask* constant is a bit mask used to extract the code from *VarType*'s return value, so that, for example,

```
VarType(V) and varTypeMask = varDouble
```

returns *True* if *V* contains a *Double* or an array of *Double*. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The *TVarData* record type defined in the *System* unit can be used to typecast variants and gain access to their internal representation. See the online Help on *VarType* for a list if codes, and note that new type codes may be added in future implementations of Object Pascal.

Variant type conversions

All integer, real, string, character, and Boolean types are assignment-compatible with *Variant*. Expressions can be explicitly cast as variants, and the *VarAsType* and *VarCast* standard routines can be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1; { integer value }
  V2 := 1234.5678; { real value }
```

```
    V3 := 'Hello world!'; { string value }
    V4 := '1000'; { string value }
    V5 := V1 + V2 + V4; { real value 2235.5678}
    I := V1; { I = 1 (integer value) }
    D := V2; { D = 1234.5678 (real value) }
    S := V3; { S = 'Hello world!' (string value) }
    I := V4; { I = 1000 (integer value) }
    S := V5; { S = '2235.5678' (string value) }
  end;
```

The compiler performs type conversions according to the following rules.

Table 5.7 Variant type conversion rules

Target Source	integer	real	string	character	Boolean
integer	converts integer formats	converts to real	converts to string representation	same as string (left)	returns *False* if 0, *True* otherwise
real	rounds to nearest integer	converts real formats	converts to string representation using regional settings	same as string (left)	returns *False* if 0, *True* otherwise
string	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/ character formats	same as string (left)	returns *False* if string is “false” (non–case-sensitive) or a numeric string that evaluates to 0, *True* if string is “true” or a nonzero numeric string; raises exception otherwise
character	same as string (above)	same as string (above)	same as string (above)	same as string-to-string	same as string (above)
Boolean	*False* = 0, *True* = –1 (255 if *Byte*)	*False* = 0, *True* = –1	*False* = “0”, *True* = “–1”	same as string (left)	*False* = *False*, *True* = *True*
Unassigned	returns 0	returns 0	returns empty string	same as string (left)	returns *False*
Null	raises exception	raises exception	raises exception	same as string (left)	raises exception

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid assignments or casts raise the *EVariantError* exception.

Special conversion rules apply to the *TDateTime* real type declared in the *System* unit. When a *TDateTime* is converted to any other type, it treated as a normal *Double*. When an integer, real, or Boolean is converted to a *TDateTime*, it is first converted to a *Double*, then read as a date-time value. When a string is converted to a *TDateTime*, it is interpreted as a date-time value using the regional settings. When an *Unassigned* value is converted to *TDateTime*, it is treated like the real or integer value 0. Converting a *Null* value to *TDateTime* raises an exception.

On Windows, if a variant references a COM interface, any attempt to convert it reads the object's default property and converts that value to the requested type. If the object has no default property, an exception is raised.

Variants in expressions

All operators except **^**, **is**, and **in** take variant operands. Operations on variants return *Variant* values; they return *Null* if one or both operands is *Null*, and raise an exception if one or both operands is *Unassigned*. In a binary operation, if only one operand is a variant, the other is converted to a variant.

The return type of an operation is determined by its operands. In general, the same rules that apply to operands of statically bound types apply to variants. For example, if *V1* and *V2* are variants that hold an integer and a real value, then `V1 + V2` returns a real-valued variant. (See "Operators" on page 4-6.) With variants, however, you can perform binary operations on combinations of values that would not be allowed using statically typed expressions. When possible, the compiler converts mismatched variants using the rules summarized in Table 5.7. For example, if *V3* and *V4* are variants that hold a numeric string and an integer, the expression `V3 + V4` returns an integer-valued variant; the numeric string is converted to an integer before the operation is performed.

Variant arrays

You cannot assign an ordinary static array to a variant. Instead, create a *variant array* by calling either of the standard functions *VarArrayCreate* or *VarArrayOf*. For example,

```
V: Variant;
 ⋮
V := VarArrayCreate([0,9], varInteger);
```

creates a variant array of integers (of length 10) and assigns it to the variant *V*. The array can be indexed using `V[0]`, `V[1]`, and so forth, but it is not possible to pass a variant array element as a **var** parameter. Variant arrays are always indexed with integers.

The second parameter in the call to *VarArrayCreate* is the type code for the array's base type. For a list of these codes, see the online Help on *VarType*. Never pass the code *varString* to *VarArrayCreate*; to create a variant array of strings, use *varOleStr*.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except *ShortString* and *AnsiString*, and if the base type of the array is *Variant*, its elements can even be heterogeneous. Use the *VarArrayRedim* function to resize a variant array. Other standard routines that operate on variant arrays include *VarArrayDimCount*, *VarArrayLowBound*, *VarArrayHighBound*, *VarArrayRef*, *VarArrayLock*, and *VarArrayUnlock*.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

OleVariant

The *OleVariant* type exists on both the Windows and Linux platforms. The main difference between *Variant* and *OleVariant* is that *Variant* can contain data types that only the current application knows what to do with. *OleVariant* can only contain the data types defined as compatible with Ole Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Pascal string, or a one of the new custom variant types) to an *OleVariant*, the runtime library tries to convert the *Variant* into one of the *OleVariant* standard data types (such as a Pascal string converts to an Ole BSTR string). For example, if a variant containing an *AnsiString* is assigned to an *OleVariant*, the *AnsiString* becomes a *WideString*. The same is true when passing a *Variant* to an *OleVariant* function parameter.

Type compatibility and identity

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include *type identity*, *type compatibility*, and *assignment-compatibility*.

Type identity

Type identity is almost straightforward. When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

T1, *T2*, *T3*, *T4*, and *Integer* all denote the same type. To create distinct types, repeat the word **type** in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called *TMyInteger* which is not identical to *Integer*.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, *TS1* and *TS2*. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```
type MyString = string[10];
var
  S1: MyString;
  S2: MyString;
```

Type compatibility

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of components.
- One is a string type and the other is a string, packed-string, or *Char* type.
- One type is *Variant* and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is *PChar* or *PWideChar* and the other is a zero-based character array of the form `array[0..`*n*`] of Char`.
- One type is *Pointer* (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the **{$T+}** compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

Assignment-compatibility

Assignment-compatibility is not a symmetric relation. An expression of type *T2* can be assigned to a variable of type *T1* if the value of the expression falls in the range of *T1* and at least one of the following conditions is satisfied.

- *T1* and *T2* are of the same type, and it is not a file type or structured type that contains a file type at any level.
- *T1* and *T2* are compatible ordinal types.
- *T1* and *T2* are both real types.
- *T1* is a real type and *T2* is an integer type.
- *T1* is *PChar* or any string type and the expression is a string constant.
- *T1* and *T2* are both string types.
- *T1* is a string type and *T2* is a *Char* or packed-string type.
- *T1* is a long string and *T2* is *PChar*.
- *T1* and *T2* are compatible packed-string types.
- *T1* and *T2* are compatible set types.
- *T1* and *T2* are compatible pointer types.
- *T1* and *T2* are both class, class-reference, or interface types and *T2* is a derived from *T1*.
- *T1* is an interface type and *T2* is a class type that implements *T1*.
- *T1* is *PChar* or *PWideChar* and *T2* is a zero-based character array of the form `array[0..`*n*`] of Char`.
- *T1* and *T2* are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type. See "Procedural types in statements and expressions" on page 5-29.)
- *T1* is *Variant* and *T2* is an integer, real, string, character, Boolean, or interface type.
- *T1* is an integer, real, string, character, or Boolean type and *T2* is *Variant*.
- *T1* is the *IUnknown* or *IDispatch* interface type and *T2* is *Variant*. (The variant's type code must be *varEmpty*, *varUnknown*, or *varDispatch* if *T1* is *IUnknown*, and *varEmpty* or *varDispatch* if *T1* is *IDispatch*.)

Declaring types

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

`type` *newTypeName* `=` *type*

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier's scope doesn't include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

X and *Y* are of the same type; at runtime, there is no way to distinguish *TValue* from *Real*. This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type information—for example, to associate a property editor with properties of a particular type—the distinction between “different name” and “different type” becomes important. In this case, use the syntax

```
type newTypeName = type type
```

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called *TValue*.

Variables

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

Declaring variables

The basic syntax for a variable declaration is

```
var identifierList: type;
```

where *identifierList* is a comma-delimited list of valid identifiers and *type* is any valid type. For example,

```
var I: Integer;
```

declares a variable *I* of type *Integer*, while

```
var X, Y: Real;
```

declares two variables—*X* and *Y*—of type *Real*.

Consecutive variable declarations do not have to repeat the reserved word **var**:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called *local*, while other variables are called *global*. Global variables can be initialized at the same time they are declared, using the syntax

```
var identifier: type = constantExpression;
```

where *constantExpression* is any constant expression representing a value of type *type*. (For more information about constant expressions, see “Constant expressions” on page 5-41.) Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
 ⋮
I := 7;
```

Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Local variables, in contrast, cannot be initialized in their declarations and contain random data until a value is assigned to them.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see Chapter 11, "Memory management".

Absolute addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive **absolute** after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

specifies that the variable *StrLen* should start at the same address as *Str*. Since the first byte of a short string contains the string's length, the value of *StrLen* is the length of *Str*.

You cannot initialize a variable in an **absolute** declaration or combine **absolute** with any other directives.

Dynamic variables

You can create dynamic variables by calling the *GetMem* or *New* procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use *FreeMem* to destroy variables created by *GetMem* and *Dispose* to destroy variables created by *New*. Other standard routines that operate on dynamic variables include *ReallocMem*, *Initialize*, *StrAlloc*, and *StrDispose*.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

Thread-local variables

Thread-local (or *thread*) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with **threadvar** instead of **var**. For example,

```
threadvar X: Integer;
```

Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the **absolute** directive.

Do not create pointer- or procedural-type thread variables, and do not use thread variables in dynamically loadable libraries (other than packages).

Dynamic variables that are ordinarily managed by the compiler—long strings, wide strings, dynamic arrays, variants, and interfaces—can be declared with **threadvar**, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory. For example,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
 ⋮
S := '';  // free the memory used by S
```

(You can free a variant by setting it to *Unassigned* and an interface or dynamic array by setting it to **nil**.)

Declared constants

Several different language constructions are referred to as "constants". There are numeric constants (also called *numerals*) like `17`, and string constants (also called *character strings* or *string literals*) like `'Hello world!'`; for information about numeric and string constants, see Chapter 4, "Syntactic elements". Every enumerated type defines constants that represent the values of that type. There are predefined constants like *True*, *False*, and **nil**. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

True constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called *MaxValue* that returns the integer 237. The syntax for declaring a true constant is

const *identifier* = *constantExpression* where *identifier* is any valid identifier and *constantExpression* is an expression that the compiler can evaluate without executing your program. (See “Constant expressions” on page 5-41 for more information.)

If *constantExpression* returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called *MyNumber*, of type *Int64*, that returns the integer 17. Otherwise, the type of the declared constant is the type of the *constantExpression*.

- If *constantExpression* is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.
- If *constantExpression* is a real, its type is *Extended*. If it is an integer, its type is given by the table below.

Table 5.8 Types for integer constants

Range of constant (hexadecimal)	Range of constant (decimal)	Type
–$8000000000000000..–$80000001	$–2^{63}$..–2147483649	*Int64*
–$80000000..–$8001	–2147483648..–32769	*Integer*
–$8000..–$81	–32768..–129	*Smallint*
–$80..–1	–128..–1	*Shortint*
0..$7F	0..127	0..127
$80..$FF	128..255	*Byte*
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	*Word*
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	*Cardinal*
$100000000..$7FFFFFFFFFFFFFFF	4294967296..2^{63}–1	*Int64*

Here are some examples of constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Constant expressions

A *constant expression* is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants *True*, *False*, and **nil**; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

Abs
Chr
Hi
High
Length
Lo
Low
Odd
Ord
Pred
Round
SizeOf
Succ
Swap
Trunc

This definition of a *constant expression* is used in several places in Object Pascal's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing **case** statements, and declaring both true and typed constants.

Examples of constant expressions:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Resource strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program. For more information, see the online Help topics on localizing applications.

Resource strings are declared like other true constants, except that the word **const** is replaced by **resourcestring**. The expression to the right of the **=** symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';        {  for explanations of format specifiers, }
  OpenError = 'Cannot open file %s';            { see 'Format strings' in the online Help }
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks\000\000';
  SomeResourceString = SomeTrueConstant;
```

The compiler automatically resolves naming conflicts among resource strings in different libraries.

Typed constants

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

In the default **{$J-}** compiler state, typed constants can not have new values assigned to them; they are, in effect, read-only variables. However, if the **{$J+}** compiler directive is in effect, typed constants can have new values assigned to them; they behave essentially like initialized variables.

Declare a typed constant like this:

```
const identifier: type = value
```

where *identifier* is any valid identifier, *type* is any type except files and variants, and *value* is an expression of type *type*. For example,

```
const Max: Integer = 100;
```

In most cases, *value* must be a constant expression; but if *type* is an array, record, procedural, or pointer type, special rules apply.

Array constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called *Digits* that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the declaration above can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

creates an array called *Maze* where

Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7

Array constants cannot contain file-type values at any level.

Record constants

To declare a record constant, specify the value of each field—as *fieldName*: *value*, with the field assignments separated by semicolons—in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record constants cannot contain file-type values at any level.

Procedural constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```
function Calc(X, Y: Integer): Integer;
begin
 ⋮
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Given these declarations, you can use the procedural constant *MyFunction* in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value **nil** to a procedural constant.

Pointer constants

When you declare a pointer constant, you must initialize it to a value that can be resolved—at least as a relative address—at compile time. There are three ways to do this: with the @ operator, with **nil**, and (if the constant is of type *PChar*) with a string literal. For example, if *I* is a global variable of type *Integer*, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a *PChar* constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```

Addresses of local (stack-allocated) and dynamic (heap-allocated) variables cannot be assigned to pointer constants.

Chapter

6

Procedures and functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A *function* is a routine that returns a value when it executes. A *procedure* is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls *SomeFunction* and assigns the result to *I*. Function calls cannot appear on the left side of an assignment statement.

Procedure calls—and, when extended syntax is enabled (**{$X+}**), function calls—can be used as complete statements. For example,

```
DoSomething;
```

calls the *DoSomething* routine; if *DoSomething* is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

Declaring procedures and functions

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the *prototype*, *heading*, or *header*. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's *body* or *block*.

The standard procedure *Exit* can occur within the body of any procedure or function. *Exit* halts execution of the routine where it occurs and immediately passes program control back to the point from which the routine was called.

Procedure declarations

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;
localDeclarations;
begin
  statements
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

- For information about the *parameterList*, see "Parameters" on page 6-11.
- For information about *directives*, see "Calling conventions" on page 6-4, "Forward and interface declarations" on page 6-5, "External declarations" on page 6-6, "Overloading procedures and functions" on page 6-8, and "Writing dynamically loadable libraries" on page 9-3. If you include more than one directive, separate them with semicolons.
- For information about *localDeclarations*, which declares local identifiers, see "Local declarations" on page 6-10.

Here is an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Given this declaration, you can call the *NumString* procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value "17" to *MyString* (which must be a **string** variable).

Within a procedure's statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like *N* and *S* in the example above); the parameter list defines a set of local variables, so don't try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

Function declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```
function functionName(parameterList): returnType; directives;
localDeclarations;
begin
  statements
end;
```

where *functionName* is any valid identifier, *returnType* is any type, *statements* is a sequence of statements that execute when the function is called, and (*parameterList*), *directives*;, and *localDeclarations*; are optional.

- For information about the *parameterList*, see "Parameters" on page 6-11.
- For information about *directives*, see "Calling conventions" on page 6-4, "Forward and interface declarations" on page 6-5, "External declarations" on page 6-6, "Overloading procedures and functions" on page 6-8, and "Writing dynamically loadable libraries" on page 9-3. If you include more than one directive, separate them with semicolons.
- For information about *localDeclarations*, which declares local identifiers, see "Local declarations" on page 6-10.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable *Result*.

For example,

```
function WF: Integer;
begin
  WF := 17;
end;
```

defines a constant function called *WF* that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
  Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
  X: Real;
  I: Integer;
begin
  X := A[0];
```

```
    for I := 1 to N - 1 do
      if X < A[I] then X := A[I];
    Max := X;
  end;
```

You can assign a value to *Result* or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to *Result* or to the function name becomes the function's return value. For example,

```
function Power(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Result and the function name always represent the same value. Hence

```
function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

returns the value 11. But *Result* is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like *Result*) to track the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. *Result*, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

As long as extended syntax is enabled (**{$X+}**), *Result* is implicitly declared in every function. Do not try to redeclare it.

If execution terminates without an assignment being made to *Result* or the function name, then the function's return value is undefined.

Calling conventions

When you declare a procedure or function, you can specify a *calling convention* using one of the directives **register**, **pascal**, **cdecl**, **stdcall**, and **safecall**. For example,

```
function MyFunction(X, Y: Real): Real; cdecl;
  ⋮
```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is **register**.

- The **register** and **pascal** conventions pass parameters from left to right; that is, the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl**, **stdcall**, and **safecall** conventions pass parameters from right to left.
- For all conventions except **cdecl**, the procedure or function removes parameters from the stack upon returning. With the **cdecl** convention, the caller removes parameters from the stack when the call returns.
- The **register** convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The **safecall** convention implements exception "firewalls." On Windows, this implements interprocess COM error notification.

The table below summarizes calling conventions.

Table 6.1 Calling conventions

Directive	Parameter order	Clean-up	Passes parameters in registers?
register	Left-to-right	Routine	Yes
pascal	Left-to-right	Routine	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Routine	No
safecall	Right-to-left	Routine	No

The default **register** convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use **register**.) The **cdecl** convention is useful when you call functions from shared libraries written in C or C++, while **stdcall** and **safecall** are recommended, in general, for calls to external code. On Windows, the operating system APIs are **stdcall** and **safecall**. Other operating systems generally use **cdecl**. (Note that **stdcall** is more efficient than **cdecl**.)

The **safecall** convention must be used for declaring dual-interface methods (see Chapter 10, "Object interfaces"). The **pascal** convention is maintained for backward compatibility. For more information on calling conventions, see Chapter 12, "Program control".

The directives **near**, **far**, and **export** refer to calling conventions in 16-bit Windows programming. They have no effect in 32-bit applications and are maintained for backward compatibility only.

Forward and interface declarations

The **forward** directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called *Calculate*. Somewhere after the **forward** declaration, the routine must be redeclared in a *defining declaration* that includes a block. The defining declaration for *Calculate* might look like this:

```
function Calculate;
 ⋮   { declarations }
begin
 ⋮   { statement block }
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the **forward** declaration exactly (except that default parameters can be omitted). If the **forward** declaration specifies an overloaded procedure or function (see "Overloading procedures and functions" on page 6-8), then the defining declaration must repeat the parameter list.

Between a **forward** declaration and its defining declaration, you can place nothing except other declarations. The defining declaration can be an **external** or **assembler** declaration, but it cannot be another **forward** declaration.

The purpose of a **forward** declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the **forward**-declared routine before it is actually defined. Besides letting you organize your code more flexibly, **forward** declarations are sometimes necessary for mutual recursions.

The **forward** directive is not allowed in the **interface** section of a unit. Procedure and function headers in the **interface** section, however, behave like **forward** declarations and must have defining declarations in the **implementation** section. A routine declared in the **interface** section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

External declarations

The **external** directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C++ function that takes a variable number of parameters, use the **varargs** directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The **varargs** directive works only with external routines and only with the **cdecl** calling convention.

Linking to object files

To call routines from a separately compiled object file, first link the object file to your application using the **$L** (or **$LINK**) compiler directive. For example,

```
On Windows: {$L BLOCK.OBJ}

On Linux:   {$L block.o}
```

links BLOCK.OBJ (Windows) or block.o (Linux) into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the *MoveWord* and *FillWord* routines from BLOCK.OBJ (Windows) or block.o (Linux).

Declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Object Pascal source code; for more information, see Chapter 13, “Inline assembler code”.

Importing functions from libraries

To import routines from a dynamically loadable library (.so or .DLL), attach a directive of the form

```
external stringConstant;
```

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Windows

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called *SomeFunction* from strlib.dll.

On Linux,

```
function SomeFunction(S: string): string; external 'strlib.so';
```

imports a function called *SomeFunction* from strlib.so.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the **external** directive:

```
external stringConstant1 name stringConstant2;
```

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine’s original name.

On Windows: For example, the following declaration imports a function from user32.dll (part of the Windows API).

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
  stdcall; external 'user32.dll' name 'MessageBoxA';
```

The function’s original name is *MessageBoxA*, but it is imported as *MessageBox*.

Instead of a name, you can use a number to identify the routine you want to import:

```
external stringConstant index integerConstant;
```

where *integerConstant* is the routine’s index in the export table.

On Linux: For example, the following declaration imports a standard system function from libc.so.6.

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
  external 'libc.so.6' name 'open';
```

The function's original name is *open*, but it is imported as *OpenFile*.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

For more information about libraries, see Chapter 9, "Libraries and packages".

Overloading procedures and functions

You can declare more than one routine in the same scope with the same name. This is called *overloading*. Overloaded routines must be declared with the **overload** directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;

function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

These declarations create two functions, both called *Divide*, that take parameters of different types. When you call *Divide*, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first *Divide* function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types—for example,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type *Extended*.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```
function Cap(S: string): string; overload;
  ⋮
procedure Cap(var Str: string); overload;
  ⋮
```

But the declarations

```
function Func(X: Real; Y: Integer): Real; overload;
 ⋮
function Func(X: Integer; Y: Real): Real; overload;
 ⋮
```

are legal.

When an overloaded routine is declared in a **forward** or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain AnsiString/PChar and WideString/WideChar parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is AnsiString/PChar.

```
procedure test(const S: String);  overload;
procedure test(const W: WideString); overload;

var
    a: string;
    b: widestring;
begin
   a := 'a';
   b := 'b';
   test(a);    // calls String version
   test(b);    // calls WideString version
   test('abc');    // calls String version
   test(WideString('abc'));   // calls widestring version
end;
```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```
procedure foo(i: integer); overload;
procedure foo(d: double);  overload;
procedure foo(v: variant); overload;
var
  v: variant;
begin
  foo(1);       // integer version
  foo(v);       // variant version
  foo(1.2);     // variant version (float literals -> extended precision)
end;
```

This example calls the variant version of foo, not the double version, because the 1.2 constant is implicitly an extended type and extended is not an exact match for double. Extended is also not an exact match for variant, but variant is considered a more general type (whereas double is a smaller type than extended).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```
const  d: double = 1.2;
  begin
    foo(d);
  end;
```

The above code works correctly, and calls the double version.

```
  const  s: single = 1.2;
  begin
    foo(s);
  end;
```

The above code also calls the double version of foo. Single is a better fit to double than to variant.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (Single, Double, Extended) along with the variant version.

If you use default parameters in overloaded routines, be careful of ambiguous parameter signatures. For more information, see “Default parameters and overloaded routines” on page 6-18.

You can limit the potential effects of overloading by qualifying a routine’s name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in *Unit1*; if no routine in *Unit1* matches the name and parameter list in the call, an error results.

For information about distributing overloaded methods in a class hierarchy, see “Overloading methods” on page 7-12. For information about exporting overloaded routines from a shared library, see “The exports clause” on page 9-5.

Local declarations

The body of a function or procedure often begins with declarations of local variables used in the routine’s statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

Nested routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called *DoSomething* contains a nested procedure.

```
procedure DoSomething(S: string);
var
  X, Y: Integer;
```

```
   procedure NestedProc(S: string);
   begin
     ⋮
   end;

begin
  ⋮
  NestedProc(S);
  ⋮
end;
```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, *NestedProc* can be called only within *DoSomething*.

For real examples of nested routines, look at the *DateTimeToString* procedure, the *ScanDate* function, and other routines in the *SysUtils* unit.

Parameters

Most procedure and function headers include a *parameter list*. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is `(X: Real; Y: Integer)`.

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the = symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by one of the reserved words **var**, **const**, and **out**. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
 ⋮
end;
```

Within the procedure or function body, the parameter names (*X* and *Y* in the first example above) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

Parameter semantics

Parameters are categorized in several ways:

- Every parameter is classified as *value*, *variable*, *constant*, or *out*. Value parameters are the default; the reserved words **var**, **const**, and **out** indicate variable, constant, and out parameters, respectively.
- Value parameters are always *typed*, while constant, variable, and out parameters can be either typed or *untyped*.
- Special rules apply to array parameters. See "Array parameters" on page 6-15.

Files and instances of structured types that contain files can be passed only as variable (**var**) parameters.

Value and variable parameters

Most parameters are either value parameters (the default) or variable (**var**) parameters. Value parameters are passed *by value*, while variable parameters are passed *by reference*. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;     // X is a value parameter
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X: Integer): Integer;   // X is a variable parameter
begin
  X := X * 2;
  Result := X;
end;
```

These functions return the same result, but only the second one—*DoubleByRef*—can change the value of a variable passed to it. Suppose we call the functions like this:

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleByValue(I);   // J = 8, I = 4
  W := DoubleByRef(V);     // W = 8, V = 8
end;
```

After this code executes, the variable *I*, which was passed to *DoubleByValue*, has the same value we initially assigned to it. But the variable *V*, which was passed to *DoubleByRef*, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more **var** parameters, no copies are made. This is illustrated in the following example.

```
procedure AddOne(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;

var I: Integer;
begin
  I := 1;
  AddOne(I, I);
end;
```

After this code executes, the value of *I* is 3.

If a routine's declaration specifies a **var** parameter, you must pass an assignable expression—that is, a variable, typed constant (in the **{$J+}** state), dereferenced pointer, field, or indexed variable—to the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in **var** parameters—for example, `DoubleByRef(MyArray[I])`—are evaluated once, before execution of the routine.

Constant parameters

A constant (**const**) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a **var** parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using **const** allows the compiler to optimize code for structured- and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the *CompareStr* function in the *SysUtils* unit:

```
function CompareStr(const S1, S2: string): Integer;
```

Because *S1* and *S2* are not modified in the body of *CompareStr*, they can be declared as constant parameters.

Out parameters

An **out** parameter, like a variable parameter, is passed by reference. With an **out** parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The **out** parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```
procedure GetInfo(out Info: SomeRecordType);
```

When you call *GetInfo*, you must pass it a variable of type *SomeRecordType*:

```
var MyRecord: SomeRecordType;
```

```
  ⋮
GetInfo(MyRecord);
```

But you’re not using *MyRecord* to pass any data to the *GetInfo* procedure; *MyRecord* is just a container where you want *GetInfo* to store the information it generates. The call to *GetInfo* immediately frees the memory used by *MyRecord*, before program control passes to the procedure.

Out parameters are frequently used with distributed-object models like COM and CORBA. In addition, you should use **out** parameters when you pass an uninitialized variable to a function or procedure.

Untyped parameters

You can omit type specifications when declaring **var**, **const**, and **out** parameters. (Value parameters must be typed.) For example,

```
procedure TakeAnything(const C);
```

declares a procedure called *TakeAnything* that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called *Equal* that compares a specified number of bytes of any two variables.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

Given the declarations

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

you could make the following calls to *Equal*:

```
Equal(Vec1, Vec2, SizeOf(TVector))            // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N)        // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5)  // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4)                        // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

String parameters

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```
procedure Check(S: string[20]);   // syntax error
```

causes a compilation error. But

```
type TString20 = string[20];
procedure Check(S: TString20);
```

is valid. The special identifier *OpenString* can be used to declare routines that take short-string parameters of varying length:

```
procedure Check(S: OpenString);
```

When the **{$H–}** and **{$P+}** compiler directives are both in effect, the reserved word **string** is equivalent to *OpenString* in parameter declarations.

Short strings, *OpenString*, **$H**, and **$P** are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

Array parameters

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer);   // syntax error
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. For most purposes, however, *open array parameters* are a better solution.

Open array parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of` *type* (rather than `array[`*X*`..`*Y*`] of` *type*) in the parameter declaration. For example,

```
function Find(A: array of Char): Integer;
```

declares a function called *Find* that takes a character array of any size and returns an integer.

Note The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The example above creates a function that takes any array of *Char* elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;
```

```
function Find(A: TDynamicCharArray): Integer;
```

For information about dynamic arrays, see "Dynamic arrays" on page 5-19.

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard *Low* and *High* functions return 0 and *Length*−1, respectively. The *SizeOf* function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped **var** parameters. They cannot be passed to *SetLength*.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a *Clear* procedure that assigns zero to each element in an array of reals and a *Sum* function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end;

function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

When you call routines that use open array parameters, you can pass *open array constructors* to them. See "Open array constructors" on page 6-19.

Variant open array parameters

Variant open array parameters allow you to pass an array of differently-typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify **array of const** as the parameter's type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called *DoSomething* that can operate on heterogeneous arrays.

The **array of const** construction is equivalent to **array of** *TVarRec*. *TVarRec*, declared in the *System* unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. *TVarRec*'s *VType* field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as *Pointer* and must be typecast to **string**. See the online Help on *TVarRec* for details.

The following example uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in *SysUtils*.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
```

We can call this function using an open array constructor (see "Open array constructors" on page 6-19). For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string "test100 T3.14159TForm".

Default parameters

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed **const** and value parameters. To provide a default value, end the parameter declaration with the **=** symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real;   // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string);   // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

the statements

```
F := Resizer;
F(N);
```

result in the values (*N*, 1.0) being passed to *Resizer*.

Default parameters are limited to values that can be specified by a constant expression. (See "Constant expressions" on page 5-41.) Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than **nil** as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

For information about calling routines with default parameter values, see "Calling procedures and functions" on page 6-19.

Default parameters and overloaded routines

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;
  ⋮
procedure Confused(I: Integer; J: Integer = 0); overload;
  ⋮
Confused(X);    //  Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

Default parameters in forward and interface declarations

If a routine has a **forward** declaration or appears in the interface section of a unit, default parameter values—if there are any—must be specified in the **forward** or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the **forward** or interface declaration exactly.

Calling procedures and functions

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called *actual parameters*, while the parameters in the routine's declaration are called *formal parameters*.

When calling a routine, remember that

- expressions used to pass typed **const** and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass **var** and **out** parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass **var** and **out** parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(,,X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();
DoSomething;
```

Open array constructors

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.

For example, given the declarations

```
var I, J: Integer;
procedure Add(A: array of Integer);
```

you could call the *Add* procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;
 ⋮
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

Open array constructors can be passed only as value or **const** parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

Chapter

7

Classes and objects

A *class*, or *class type*, defines a structure consisting of *fields*, *methods*, and *properties*. Instances of a class type are called *objects*. The fields, methods, and properties of a class are called its *components* or *members*.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class's fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects—that is, instances of a class. Some methods (called *class methods*) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have *access specifiers*, which determine how their data are read and modified. From other parts of a program—outside of the object itself—a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called *constructors* and *destructors*.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value **nil**. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the *Size* property of the object referenced by *SomeObject*; you would *not* write this as `SomeObject^.Size := 100`.

Class types

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type className = class (ancestorClass)
  memberList
 end;
```

where *className* is any valid identifier, (*ancestorClass*) is optional, and *memberList* declares members—that is, fields, methods, and properties—of the class. If you omit (*ancestorClass*), then the new class inherits directly from the predefined *TObject* class. If you include (*ancestorClass*) and *memberList* is empty, you can omit `end`. A class type declaration can also include a list of *interfaces* implemented by the class; see "Implementing interfaces" on page 10-4.

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the *TMemoryStream* class from the *Classes* unit.

```
type
TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
  end;
```

TMemoryStream descends from *TStream* (in the *Classes* unit), inheriting most of its members. But it defines—or redefines—several methods and properties, including its destructor method, *Destroy*. Its constructor, *Create*, is inherited without change from *TObject*, and so is not redeclared. Each member is declared as *private*, *protected*, or *public* (this class has no *published* members); for explanations of these terms, see "Visibility of class members" on page 7-4.

Given this declaration, you can create an instance of *TMemoryStream* as follows:

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

Inheritance and scope

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called *TSomeControl* that descends from *TControl*. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence *TSomeControl* contains all of the members defined in *TControl* and in each of *TControl*'s ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

TObject and TClass

The *TObject* class, declared in the *System* unit, is the ultimate ancestor of all other classes. *TObject* defines only a handful of methods, including a basic constructor and destructor. In addition to *TObject*, the *System* unit declares the class-reference type *TClass*:

```
TClass = class of TObject;
```

For more information about *TObject*, see the online help. For more information about class-reference types, see "Class references" on page 7-23.

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from *TObject*. Thus

```
type TMyClass = class
  ⋮
end;
```

is equivalent to

```
type TMyClass = class(TObject)
  ⋮
end;
```

The latter form is recommended for readability.

Compatibility of class types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TSquare = class(TRectangle);
var
  Fig: TFigure;
```

the variable *Fig* can be assigned values of type *TFigure*, *TRectangle*, and *TSquare*.

Object types

As an alternative to class types, you can declare *object types* using the syntax

```
type objectTypeName = object (ancestorObjectType)
  memberList
end;
```

where *objectTypeName* is any valid identifier, (*ancestorObjectType*) is optional, and *memberList* declares fields, methods, and properties. If (*ancestorObjectType*) is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from *TObject*, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the *New* procedure and destroy them with the *Dispose* procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended.

Visibility of class members

Every member of a class has an attribute called *visibility*, which is indicated by one of the reserved words **private**, **protected**, **public**, **published**, or **automated**. For example,

```
published property Color: TColor read GetColor write SetColor;
```

declares a published property called *Color*. Visibility determines where and how a member can be accessed, with private representing the least accessibility, protected representing an intermediate level of accessibility, and public, published, and automated representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default published, provided the class is compiled in the **{$M+}** state or is derived from a class compiled in the **{$M+}** state; otherwise, such members are public.

For readability, it is best to organize a class declaration by visibility, placing all the private members together, followed by all the protected members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new "section" of the declaration. So a typical class declaration should like this:

```
type
  TMyClass = class(TControl)
  private
   ⋮ { private declarations here}
  protected
   ⋮ { protected declarations here }
  public
   ⋮ { public declarations here }
  published
   ⋮ { published declarations here }
  end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a protected property can be made public in a descendant, but not private. Moreover, published members cannot become public in a descendant class. For more information, see “Property overrides and redeclarations” on page 7-22.

Private, protected, and public members

A *private* member is invisible outside of the unit or program where its class is declared. In other words, a private method cannot be called from another module, and a private field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another’s private members without making those members more widely accessible.

A *protected* member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. In other words, a protected method can be called, and a protected field or property read or written to, from the definition of any method belonging to a class that descends from the one where the protected member is declared. Members that are intended for use only in the implementation of derived classes are usually protected.

A *public* member is visible wherever its class can be referenced.

Published members

Published members have the same visibility as public members. The difference is that *runtime type information* (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the Object Inspector, and to associate specific methods (called *event handlers*) with specific properties (called *events*).

Published properties are restricted to certain data types. Ordinal, string, class, interface, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except *Real48* can be published. Properties of an array type (as distinct from *array properties*, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties of all publishable types (see “Array properties” on page 7-19), and properties of enumerated types that include anonymous values (see “Enumerated types with explicitly assigned ordinality” on page 5-7). If you publish a property of this kind, the Object Inspector won’t display it correctly, nor will the property’s value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the **{$M+}** state or descends from a class compiled in the **{$M+}** state. Most classes with published members derive from *TPersistent*, which is compiled in the **{$M+}** state, so it is seldom necessary to use the **$M** directive.

Automated members

Automated members have the same visibility as public members. The difference is that *Automation type information* (required for Automation servers) is generated for automated members. Automated members typically appear only in Windows classes and is not recommended for Linux programming. The **automated** reserved word is maintained for backward compatibility. The *TAutoObject* class in the *ComObj* unit does not use **automated**.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool*, and all interface types.
- Method declarations must use the default **register** calling convention. They can be virtual, but not dynamic.
- Property declarations can include access specifiers (**read** and **write**) but other specifiers (**index**, **stored**, **default**, and **nodefault**) are not allowed. Access specifiers must list a method identifier that uses the default **register** calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a **dispid** directive. Specifying an already used ID in a **dispid** directive causes an error.

On Windows, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Windows only), see “Automation objects (Windows only)” on page 10-10.

Forward declarations and mutually dependent classes

If the declaration of a class type ends with the word **class** and a semicolon—that is, if it has the form

```
type className = class;
```

with no ancestor or class members listed after the word **class**—then it is a *forward declaration*. A forward declaration must be resolved by a *defining declaration* of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```
type
  TFigure = class;  // forward declaration
  TDrawing = class
    Figure: TFigure;
     ⋮
  end;

  TFigure = class  // defining declaration
    Drawing: TDrawing;
     ⋮
  end;
```

Do not confuse forward declarations with complete declarations of types that derive from *TObject* without declaring any class members.

```
type
  TFirstClass = class;           // this is a forward declaration

  TSecondClass = class           // this is a complete class declaration
    end;                         //

  TThirdClass = class(TObject);  // this is a complete class declaration
```

Fields

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. All field declarations must occur before any property or method declarations. For example, the following declaration creates a class called *TNumber* whose only member, other than the methods is inherits from *TObject*, is an integer field called *Int*.

```
type TNumber = class
  Int: Integer;
end;
```

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```
type
  TAncestor = class
    Value: Integer;
  end;

  TDescendant = class(TAncestor)
    Value: string;  // hides the inherited Value field
  end;

var
  MyObject: TAncestor;

begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!';  // error
  TDescendant(MyObject).Value := 'Hello!';  // works!
end;
```

Although *MyObject* holds an instance of *TDescendant*, it is declared as *TAncestor*. The compiler therefore interprets *MyObject.Value* as referring to the (integer) field declared in *TAncestor*. Both fields, however, exist in the *TDescendant* object; the inherited *Value* is hidden by the new one, and can be accessed through a typecast.

Methods

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example,

```
SomeObject.Free
```

calls the *Free* method in *SomeObject*.

Method declarations and implementations

Within a class declaration, methods appear as procedure and function headings, which work like **forward** declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of *TMyClass* includes a method called *DoSomething*:

```
type
  TMyClass = class(TObject)
     ⋮
    procedure DoSomething;
     ⋮
  end;
```

A defining declaration for *DoSomething* must occur later in the module:

```
procedure TMyClass.DoSomething;
begin
 ⋮
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class's methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does so, the order, type, and names of the parameters must match exactly, and, if the method is a function, so must the return value.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

reintroduce; **overload**; *binding*; *calling convention*; **abstract**; *warning* where *binding* is **virtual**, **dynamic**, or **override**; *calling convention* is **register**, **pascal**, **cdecl**, **stdcall**, or **safecall**; and *warning* is **platform**, **deprecated**, or **library**.

Inherited

The reserved word **inherited** plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If **inherited** is followed by the name of a member, it represents a normal method call or reference to a property or field—except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited *Create*.

When **inherited** has no identifier after it, it refers to the inherited method with the same name as the enclosing method. In this case, **inherited** takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

Self

Within the implementation of a method, the identifier *Self* references the object in which the method is called. For example, here is the implementation of *TCollection*'s *Add* method in the *Classes* unit.

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

The *Add* method calls the *Create* method in the class referenced by the *FItemClass* field, which is always a *TCollectionItem* descendant. *TCollectionItem.Create* takes a single parameter of type *TCollection*, so *Add* passes it the *TCollection* instance object where *Add* is called. This is illustrated in the following code.

```
var MyCollection: TCollection;
 ⋮
MyCollection.Add  // MyCollection is passed to the TCollectionItem.Create method
```

Self is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class's methods. In this case, you can access the original member identifier as `Self`.*Identifier*.

For information about *Self* in class methods, see "Class methods" on page 7-25.

Method binding

Methods can be *static* (the default), *virtual*, or *dynamic*. Virtual and dynamic methods can be *overridden*, and they can be *abstract*. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

Static methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the *Draw* methods are static.

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to *Figure.Draw*, the *Figure* variable references an object of class *TRectangle*, but the call invokes the implementation of *Draw* in *TFigure*, because the declared type of the *Figure* variable is *TFigure*.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;  // calls TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;  // calls TFigure.Draw
  TRectangle(Figure).Draw;  // calls TRectangle.Draw
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;  // calls TRectangle.Draw
  Rectangle.Destroy;
end;
```

Virtual and dynamic methods

To make a method virtual or dynamic, include the **virtual** or **dynamic** directive in its declaration. Virtual and dynamic methods, unlike static methods, can be *overridden* in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method call—not the declared type of the variable—determines which implementation to activate.

To override a method, redeclare it with the **override** directive. An **override** declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the *Draw* method declared in *TFigure* is overridden in two descendant classes.

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;  // calls TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;  // calls TEllipse.Draw
  Figure.Destroy;
end;
```

Only virtual and dynamic methods can be overridden. All methods, however, can be *overloaded*; see “Overloading methods”.

Virtual versus dynamic

Virtual and dynamic methods are semantically equivalent. They differ only in the implementation of method-call dispatching at runtime. Virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

Overriding versus hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn’t include **override**, the new declaration merely *hides* the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```
type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act;  // Act is redeclared, but not overridden
  end;
```

```
var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act;  // calls T1.Act
end;
```

Reintroduce

The **reintroduce** directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```
procedure DoSomething; reintroduce;  // the ancestor class also has a DoSomething method
```

Use **reintroduce** when you want to hide an inherited virtual method with a new one.

Abstract methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive **abstract** after **virtual** or **dynamic**. For example,

```
procedure DoSomething; virtual; abstract;
```

You can call an abstract method only in a class or instance of a class in which the method has been overridden.

Overloading methods

A method can be redeclared using the **overload** directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the **reintroduce** directive when you redeclare it in descendant classes. For example,

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;
  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
   ⋮
  SomeObject := T2.Create;
  SomeObject.Test('Hello!');  // calls T2.Test
  SomeObject.Test(7);         // calls T1.Test
```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer  // error
     ⋮
```

Methods that serve as property **read** or **write** specifiers cannot be overloaded.

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see “Overloading procedures and functions” on page 6-8.

Constructors

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word **constructor**. Examples:

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Constructors must use the default **register** calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor *Create*.

To create an object, call the constructor method in a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object on the heap, sets the values of all ordinal fields to zero, assigns **nil** to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the *Destroy* destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor’s implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word **inherited** to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);  // Initialize inherited parts
  Width := 65;  // Change inherited properties
  Height := 65;
  FPen := TPen.Create;  // Initialize new fields
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), **nil** (pointer and class types), empty (string types), or *Unassigned* (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as **virtual** is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects—that is, construction of objects whose types aren't known at compile time. (See "Class references" on page 7-23.)

Destructors

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word **destructor**. Examples:

```
destructor Destroy;
destructor Destroy; override;
```

Destructors must use the default **register** calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited *Destroy* method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, *Destroy* is automatically called to dispose of the unfinished object. This means that *Destroy* must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always **nil**. A destructor should therefore check for **nil** values before operating on class-type or pointer-type fields. Calling the *Free* method (defined in *TObject*), rather than *Destroy*, offers a convenient way of checking for **nil** values before destroying an object.

Message methods

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. The VCL uses message methods to respond to Windows messages. CLX does not use message methods to respond to system events.

A message method is created by including the **message** directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Windows message IDs defined, along with corresponding record types, in the *Messages* unit. A message method must be a procedure that takes a single **var** parameter.

For example, on Windows:

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ⋮
  end;
```

For example, on Linux or for cross-platform programming, you would handle messages as follows:

```
const
  ID_REFRESH = $0001;

type
  TTextBox = class(TCustomControl)
  private
    procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
    ⋮
  end;
```

A message method does not have to include the **override** directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

Implementing message methods

The implementation of a message method can call the inherited message method, as in this example (for Windows):

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

On Linux or for cross-platform programming, you would write the same example as follows:

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
  else
    inherited;
end;
```

The **inherited** statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, **inherited** calls the *DefaultHandler* method originally defined in *TObject*.

The implementation of *DefaultHandler* in *TObject* simply returns without performing any actions. By overriding *DefaultHandler*, a class can implement its own default handling of messages. On Windows, the *DefaultHandler* method for VCL controls calls the Windows *DefWindowProc* function.

Message dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the *Dispatch* method inherited from *TObject*:

```
procedure Dispatch(var Message);
```

The *Message* parameter passed to *Dispatch* must be a record whose first entry is a field of type *Cardinal* containing a message ID.

Dispatch searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, *Dispatch* calls *DefaultHandler*.

Properties

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

`property` *propertyName*`[`*indexes*`]:` *type* `index` *integerConstant specifiers*`;`

where

- *propertyName* is any valid identifier.
- `[`*indexes*`]` is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier*$_1$`,` ...`,` *identifier*$_n$`:` *type*. For more information, see "Array properties" on page 7-19.
- *type* must be a predefined or previously declared data type. That is, property declarations like **`property`** `Num: 0..9 ...` are invalid.
- the `index` *integerConstant* clause is optional. For more information, see "Index specifiers" on page 7-20.
- *specifiers* is a sequence of **read**, **write**, **stored**, **default** (or **nodefault**), and **implements** specifiers. Every property declaration must have at least one **read** or **write** specifier. (For information about **implements**, see "Implementing interfaces by delegation" on page 10-6.)

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as **var** parameters, nor can the @ operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a **read** method that retrieves a value from a database or generates a random value.

Property access

Every property has a **read** specifier, a **write** specifier, or both. These are called *access specifiers* and they have the form

`read` *fieldOrMethod*
`write` *fieldOrMethod*

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.
- If *fieldOrMethod* is a method, it cannot be overloaded. Moreover, access methods for a published property must use the default **register** calling convention.
- In a **read** specifier, if *fieldOrMethod* is a method, it must be a parameterless function whose result type is the same as the property's type.
- In a **write** specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or **const** parameter of the same type as the property.

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the *GetColor* method must be declared as

```
function GetColor: TColor;
```

and the *SetColor* method must be declared as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(The name of *SetColor*'s parameter, of course, doesn't have to be *Value*.)

When a property is referenced in an expression, its value is read using the field or method listed in the **read** specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the **write** specifier.

The example below declares a class called *TCompass* with a published property called *Heading*. The value of *Heading* is read through the *FHeading* field and written through the *SetHeading* procedure.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    ⋮
  end;
```

Given this declaration, the statements

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the *TCompass* class, no action is associated with reading the *Heading* property; the **read** operation consists of retrieving the value stored in the *FHeading* field. On the other hand, assigning a value to the *Heading* property translates into a call to the *SetHeading* method, which, presumably, stores the new value in the *FHeading* field as well as performing other actions. For example, *SetHeading* might be implemented like this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;  // update user interface to reflect new value
  end;
end;
```

A property whose declaration includes only a **read** specifier is a *read-only* property, and one whose declaration includes only a **write** specifier is a *write-only* property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

Array properties

Array properties are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a **read** specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a **write** specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or **const** parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by indexing the property identifier. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

On Linux, you would use a path such as '/usr/local/bin' in place of 'C:\DELPHI\BIN' in the above example.

The definition of an array property can be followed by the **default** directive, in which case the array property becomes the default property of the class. For example,

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ⋮
  end;
```

If a class has a default property, you can access that property with the abbreviation *object*[*index*], which is equivalent to *object*.*property*[*index*]. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always determines an object's default property statically.

Index specifiers

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive **index** followed by an integer constant between –2147483647 and 2147483647. If a property has an index specifier, its **read** and **write** specifiers must list methods rather than fields. For example,

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
    property Left: Longint index 0 read GetCoordinate write SetCoordinate;
    property Top: Longint index 1 read GetCoordinate write SetCoordinate;
    property Right: Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
    ⋮
  end;
```

An access method for a property with an index specifier must take an extra value parameter of type *Integer*. For a **read** function, it must be the last parameter; for a **write** procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if *Rectangle* is of type *TRectangle*, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage specifiers

The optional **stored**, **default**, and **nodefault** directives are called *storage specifiers*. They have no effect on program behavior, but control the way runtime type information (RTTI) is maintained. Specifically, storage specifiers determine whether or not to save the values of published properties in form files.

The **stored** directive must be followed by *True*, *False*, the name of a *Boolean* field, or the name of a parameterless method that returns a *Boolean* value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no **stored** directive, it is treated as if `stored True` were specified.

The **default** directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited **default** value without specifying a new one, use the **nodefault** directive. The **default** and **nodefault** directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without **default** or **nodefault**, it is treated as if **nodefault** were specified. For reals, pointers, and strings, there is an implicit **default** value of 0, **nil**, and `''` (the empty string), respectively.

When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its **default** value (or if there is no **default** value) and the **stored** specifier is *True*, then the property's value is saved. Otherwise, the property's value is not saved.

Note Storage specifiers are not supported for array properties. The **default** directive has a different meaning when used in an array property declaration. See "Array properties" on page 7-19.

Property overrides and redeclarations

A property declaration that doesn't specify a type is called a *property override*. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word **property** followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as protected, a derived class can redeclare it in a public or published section of the class. Property overrides can include **read**, **write**, **stored**, **default**, and **nodefault** directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an **implements** directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```
type
  TAncestor = class
   ⋮
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    ⋮
  end;
type
  TDerived = class(TAncestor)
   ⋮
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    ⋮
  end;
```

The override of *Size* adds a **write** specifier to allow the property to be modified. The overrides of *Text* and *Color* change the visibility of the properties from protected to published. The property override of *Color* also specifies that the property should be filed if its value isn't *clBlue*.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always *static*. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to *MyObject.Value* invokes *Method1* or *Method2*, even though *MyObject* holds an instance of *TDescendant*. But you can cast *MyObject* to *TDescendant* to access the descendant class's properties and their access specifiers.

```
type
  TAncestor = class
     ⋮
    property Value: Integer read Method1 write Method2;
  end;

  TDescendant = class(TAncestor)
     ⋮
    property Value: Integer read Method3 write Method4;
  end;

var MyObject: TAncestor;
 ⋮
MyObject := TDescendant.Create;
```

Class references

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

Class-reference types

A class-reference type, sometimes called a *metaclass*, is denoted by a construction of the form

`class of` *type*

where *type* is any class type. The identifier *type* itself denotes a value whose type is `class of` *type*. If *type*$_1$ is an ancestor of *type*$_2$, then `class of` *type*$_2$ is assignment-compatible with `class of` *type*$_1$. Thus

```
type TClass = class of TObject;
var AnyObj: TClass;
```

declares a variable called *AnyObj* that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value **nil** to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for *TCollection* (in the *Classes* unit):

```
type TCollectionItemClass = class of TCollectionItem;
  ⋮
constructor Create(ItemClass: TCollectionItemClass);
```

This declaration says that to create a *TCollection* instance object, you must pass to the constructor the name of a class descending from *TCollectionItem*.

Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

Constructors and class references

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

The *CreateControl* function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to *CreateControl* can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

Class operators

Every class inherits from *TObject* methods called *ClassType* and *ClassParent* that return, respectively, a reference to the class of an object and of an object's immediate ancestor. Both methods return a value of type *TClass* (where `TClass = class of TObject`), which can be cast to a more specific type. Every class also inherits a method called *InheritsFrom* that tests whether the object where it is called descends from a specified class. These methods are used by the **is** and **as** operators, and it is seldom necessary to call them directly.

The is operator

The **is** operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression

object `is` *class*

returns *True* if *object* is an instance of the class denoted by *class* or one of its descendants, and *False* otherwise. (If *object* is **nil**, the result is *False*.) If the declared type of *object* is unrelated to *class*—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to *TEdit* after first verifying that the object it references is an instance of *TEdit* or one of its descendants.

The as operator

The **as** operator performs checked typecasts. The expression

```
   object as class
```

returns a reference to the same object as *object*, but with the type given by *class*. At runtime, *object* must be an instance of the class denoted by *class* or one of its descendants, or be **nil**; otherwise an exception is raised. If the declared type of *object* is unrelated to *class*—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results. For example,

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

The rules of operator precedence often require **as** typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```

Class methods

A class method is a method (other than a constructor) that operates on classes instead of objects. The definition of a class method must begin with the reserved word **class**. For example,

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ⋮
  end;
```

The defining declaration of a class method must also begin with **class**. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  ⋮
end;
```

In the defining declaration of a class method, the identifier *Self* represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class *C*, then *Self* is of the type `class of` *C*. Thus you cannot use *Self* to access fields, properties, and normal (object) methods, but you can use it to call constructors and other class methods.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of *Self*.

Exceptions

An *exception* is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an *exception handler*, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the *SysUtils* unit, all runtime errors are automatically converted into exceptions. Errors that would otherwise terminate an application—such as insufficient memory, division by zero, and general protection faults—can be caught and handled.

When to use exceptions

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The complexity of Object Pascal's exception-handling mechanism, however, makes it inefficient, and it should therefore be used judiciously. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a **try...except** or **try...finally** statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in **if...then** statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
  AssignFile(F, FileName);
  Reset(F);  // raises an EInOutError exception if file is not found
except
  on Exception do ...
end;
```

But you could also avoid the overhead of exception handling by using

```
if FileExists(FileName) then  // returns False if file is not found; raises no exception
begin
```

```
    AssignFile(F, FileName);
    Reset(F);
  end;
```

Assertions provide another way of testing a Boolean condition anywhere in your source code. When an *Assert* statement fails, the program either halts or (if it uses the *SysUtils* unit) raises an *EAssertionFailed* exception. Assertions should be used only to test for conditions that you do not expect to occur. For more information, see the online Help for the standard procedure *Assert*.

Declaring exception types

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the *Exception* class defined in *SysUtils*.

You can group exceptions into families using inheritance. For example, the following declarations in *SysUtils* define a family of exception types for math errors.

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

Given these declarations, you can define a single *EMathError* exception handler that also handles *EInvalidOp*, *EZeroDivide*, *EOverflow*, and *EUnderflow*.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

Raising and handling exceptions

To create an exception object, call the exception class's constructor within a **raise** statement. For example,

```
raise EMathError.Create;
```

In general, the form of a **raise** statement is

`raise` *object* `at` *address*

where *object* and `at` *address* are both optional. If *object* is omitted, the statement re-raises the current exception; see "Re-raising exceptions" on page 7-30. When an *address* is specified, it is usually a pointer to a procedure or function; use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is *raised*—that is, referenced in a **raise** statement—it is governed by special exception-handling logic. A **raise** statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose **try...except** block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an *ERangeError* exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);  // StrToInt is declared in SysUtils
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

Notice the *CreateFmt* method called in the **raise** statement. *Exception* and its descendants have special constructors that provide alternative ways to create exception messages and context IDs. See the online Help for details.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

Note Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the *SysUtils* unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units—including *SysUtils*—are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program.

Try...except statements

Exceptions are handled within **try...except** statements. For example,

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide *Y* by *Z*, but calls a routine named *HandleZeroDivide* if an *EZeroDivide* exception is raised.

The syntax of a **try...except** statement is

`try` *statements* `except` *exceptionBlock* `end`

where *statements* is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

  `else` *statements*

An exception handler has the form

on *identifier*: *type* do *statement*

where *identifier*: is optional (if included, *identifier* can be any valid identifier), *type* is a type used to represent exceptions, and *statement* is any statement.

A **try...except** statement executes the statements in the initial *statements* list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial *statements* list, either by a **raise** statement in the *statements* list or by a procedure or function called from the *statements* list, an attempt is made to "handle" the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler "matches" an exception just in case the *type* in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the *statement* in the **else** clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered **try...except** statement that has not yet exited. If no appropriate handler, **else** clause, or statement list is found there, the search propagates to the next-most-recently entered **try...except** statement, and so forth. If the outermost **try...except** statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the **try...except** statement where the handling occurs, and control is transferred to the executed exception handler, **else** clause, or statement list. This process discards all procedure and function calls that occurred after entering the **try...except** statement where the exception is handled. The exception object is then automatically destroyed through a call to its *Destroy* destructor and control is passed to the statement following the **try...except** statement. (If a call to the *Exit*, *Break*, or *Continue* standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. *EMathError* appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows **on...do**. The scope of the identifier is limited to that statement. For example,

```
try
 ⋮
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an **else** clause, the **else** clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** clause handles any exception that isn't an *EMathError*.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try
 ⋮
except
  HandleException;
end;
```

Here, the *HandleException* routine handles any exception that occurs as a result of executing the statements between **try** and **except**.

Re-raising exceptions

When the reserved word **raise** occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then *re-raise* the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the *GetFileList* function allocates a *TStringList* object and fills it with file names matching a specified search path:

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
```

```
      while I = 0 do
      begin
        Result.Add(SearchRec.Name);
        I := FindNext(SearchRec);
      end;
    except
      Result.Free;
      raise;
    end;
  end;
```

GetFileList creates a *TStringList* object, then uses the *FindFirst* and *FindNext* functions (defined in *SysUtils*) to initialize it. If the initialization fails—for example because the search path is invalid, or because there is not enough memory to fill in the string list—*GetFileList* needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a **try...except** statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

Nested exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the *Tan* function below.

```
type
  ETrigError = class(EMathError);

function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
  end;
end;
```

If an *EMathError* exception occurs during execution of *Tan*, the exception handler raises an *ETrigError*. Since *Tan* does not provide a handler for *ETrigError*, the exception propagates beyond the original exception handler, causing the *EMathError* exception to be destroyed. To the caller, it appears as if the *Tan* function has raised an *ETrigError* exception.

Try...finally statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a **try...finally** statement.

The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```
Reset(F);
try
 ⋮ // process file F
finally
  CloseFile(F);
end;
```

The syntax of a **try...finally** statement is

try *statementList*$_1$ finally *statementList*$_2$ end

where each *statementList* is a sequence of statements delimited by semicolons. The **try...finally** statement executes the statements in *statementList*$_1$ (the **try** clause). If *statementList*$_1$ finishes without raising exceptions, *statementList*$_2$ (the **finally** clause) is executed. If an exception is raised during execution of *statementList*$_1$, control is transferred to *statementList*$_2$; once *statementList*$_2$ finishes executing, the exception is re-raised. If a call to the *Exit*, *Break*, or *Continue* procedure causes control to leave *statementList*$_1$, *statementList*$_2$ is automatically executed. Thus the **finally** clause is always executed, regardless of how the **try** clause terminates.

If an exception is raised but not handled in the **finally** clause, that exception is propagated out of the **try...finally** statement, and any exception already raised in the **try** clause is lost. The **finally** clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

Standard exception classes and routines

The *SysUtils* unit declares several standard routines for handling exceptions, including *ExceptObject*, *ExceptAddr*, and *ShowException*. *SysUtils* and other units also include dozens of exception classes, all of which (aside from *OutlineError*) derive from *Exception*.

The *Exception* class has properties called *Message* and *HelpContext* that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways. See the online Help for details.

Chapter

8

Standard routines and I/O

This chapter discusses text and file I/O and summarizes standard library routines. Many of the procedures and functions listed here are defined in the *System* unit, which is implicitly compiled with every application. Others are built into the compiler but are treated as if they were in the *System* unit.

Some standard routines are in units such as *SysUtils*, which must be listed in a **uses** clause to make them available in programs. You cannot, however, list *System* in a **uses** clause, nor should you modify the *System* unit or try to rebuild it explicitly.

For more information about the routines listed here, see the online Help.

File input and output

The table below lists input and output routines.

Table 8.1 Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.

Table 8.1 Input and output procedures and functions (continued)

Procedure or function	Description
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what *Read* does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as *Write*, and then writes an end-of-line marker to the text file.

A file variable is any variable whose type is a file type. There are three classes of file: *typed*, *text*, and *untyped*. The syntax for declaring file types is given in “File types” on page 5-24.

Before a file variable can be used, it must be associated with an external file through a call to the *AssignFile* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be “opened” to prepare it for input or output. An existing file can be opened via the *Reset* procedure, and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write*, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure *CloseFile*. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{$I+}** and **{$I–}** compiler directives. When I/O checking is off—that is, when a procedure or function call is compiled in the **{$I–}** state—an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function *IOResult* instead.

You must call the *IOResult* function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{$I+}** is the current state, the next I/O function call will fail with the lingering *IOResult* error.

Text files

This section summarizes I/O using file variables of the standard type *Text*.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character). The type *Text* is distinct from the type `file of Char`.

For text files, there are special forms of *Read* and *Write* that let you read and write values that are not of type *Char*. Such values are automatically translated to and from their character representation. For example, `Read(F, I)`, where *I* is a type *Integer* variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in *I*.

There are two standard text-file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable *Output* is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, *Input* and *Output* are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Note Text-oriented I/O is available only in console applications—that is, applications compiled with the “Generate console application” option checked on the Linker page of the Project Options dialog box or with the **-cc** command-line compiler option. In a GUI (non-console) application, any attempt to read or write using *Input* or *Output* will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, `Read(X)` corresponds to `Read(Input, X)` and `Write(X)` corresponds to `Write(Output, X)`.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using *AssignFile*, and opened using *Reset*, *Rewrite*, or *Append*. An exception is raised if you pass a file that was opened with *Reset* to an output-oriented procedure or function. An exception is also raised if you pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

Untyped files

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word **file** and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the *Reset* and *Rewrite* procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for *Read* and *Write*, all typed-file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

Text-file device drivers

You can define your own text-file device drivers for your programs. A text-file device driver is a set of four functions that completely implement an interface between Object Pascal's file system and some device.

The four functions that define each device driver are *Open*, *InOut*, *Flush*, and *Close*. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where *DeviceFunc* is the name of the function (that is, *Open*, *InOut*, *Flush*, or *Close*). For information about the *TTextRec* type, see the online Help. The return value of a device-interface function becomes the value returned by *IOResult*. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized *Assign* procedure. The *Assign* procedure must assign the addresses of the four device-interface functions to the four function pointers in the text-file variable. In addition, it should store the *fmClosed* "magic" constant in the *Mode* field, store the size of the text-file buffer in *BufSize*, store a pointer to the text-file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device-interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device-interface functions can use the *UserData* field in the file record to store private information. This field isn't modified by the product file system at any time.

Device functions

The functions that make up a text-file device driver are described below.

The Open function

The *Open* function is called by the *Reset*, *Rewrite*, and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput*, *fmOutput*, or *fmInOut* to indicate whether the *Open* function was called from *Reset*, *Rewrite*, or *Append*.

The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutput* before *Open* returns.

Open is always called before any of the other device-interface functions. For that reason, *AssignDev* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode*, *Open* can then install pointers to either input- or output-oriented functions. This saves the *InOut*, *Flush* functions and the *CloseFile* procedure from determining the current mode.

The InOut function

The *InOut* function is called by the *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln*, and *CloseFile* standard routines whenever input or output from the device is required.

When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into `BufPtr^`, and returns the number of characters read in *BufEnd*. In addition, it stores zero in *BufPos*. If the *InOut* function returns zero in *BufEnd* as a result of an input request, *Eof* becomes *True* for the file.

When *Mode* is *fmOutput*, the *InOut* function writes *BufPos* characters from `BufPtr^`, and returns zero in *BufPos*.

The Flush function

The *Flush* function is called at the end of each *Read*, *Readln*, *Write*, and *Writeln*. It can optionally flush the text-file buffer.

If *Mode* is *fmInput*, the *Flush* function can store zero in *BufPos* and *BufEnd* to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The *Close* function is called by the *CloseFile* standard procedure to close a text file associated with a device. (The *Reset*, *Rewrite*, and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutput*, then before calling *Close*, the file system calls the *InOut* function to ensure that all characters have been written to the device.

Handling null-terminated strings

Object Pascal's extended syntax allows the *Read*, *Readln*, *Str*, and *Val* standard procedures to be applied to zero-based character arrays, and allows the *Write*, *Writeln*, *Val*, *AssignFile*, and *Rename* standard procedures to be applied to both zero-based character arrays and character pointers. In addition, the following functions are provided for handling null-terminated strings. For more information about null-terminated strings, see "Working with null-terminated strings" on page 5-13.

Table 8.2 Null-terminated string functions

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using *StrAlloc* or *StrNew*.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using *StrAlloc* or *StrNew*.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.

Table 8.2 Null-terminated string functions (continued)

Function	Description
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase.
StrMove	Moves a block of characters from one string to another.
StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to uppercase.

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with *Ansi-*. For example, the multibyte version of *StrPos* is *AnsiStrPos*. Multibyte character support is operating-system dependent and based on the current locale.

Wide-character strings

The *System* unit provides three functions, *WideCharToString*, *WideCharLenToString*, and *StringToWideChar*, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

For more information about wide-character strings, see “About extended character sets” on page 5-13.

Other standard routines

The table below lists frequently used procedures and functions found in Borland product libraries. This is not an exhaustive inventory of standard routines. For more information about these and other routines, see the online Help.

Table 8.3 Other standard routines

Procedure or function	Description
Abort	Ends the process without reporting an error.
Addr	Returns a pointer to a specified object.
AllocMem	Allocates a memory block and initializes each byte to zero.
ArcTan	Calculates the arctangent of the given number.
Assert	Tests whether a boolean expression is *True*.
Assigned	Tests for a **nil** (unassigned) pointer or procedural variable.
Beep	Generates a standard beep using the computer speaker.

Table 8.3 Other standard routines (continued)

Procedure or function	Description
Break	Causes control to exit a **for**, **while**, or **repeat** statement.
ByteToCharIndex	Returns the position of the character containing a specified byte in a string.
Chr	Returns the character for a specified value.
Close	Terminates the association between a file variable and an external file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of **for**, **while**, or **repeat** statements.
Copy	Returns a substring of a string or a segment of a dynamic array.
Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type *TDateTime* to a string.
DateToStr	Converts a variable of type *TDateTime* to a string.
Dec	Decrements an ordinal variable.
Dispose	Releases memory allocated for a dynamic variable.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.
Finalize	Uninitializes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Disposes of a dynamic variable.
GetMem	Creates a dynamic variable and a pointer to the address of the block.
GetParentForm	Returns the form or property page that contains a specified control.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.
High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.
Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.

Table 8.3 Other standard routines (continued)

Procedure or function	**Description**
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a new dynamic variable and references it with a specified pointer.
Now	Returns the current date and time.
Ord	Returns the ordinal value of an ordinal-type expression.
Pos	Returns the index of the first character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a specified address to a pointer.
Random	Generates random numbers within a specified range.
ReallocMem	Reallocates a dynamic variable.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.
ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Formats a string and returns it to a variable.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (*TDateTime*).
StrToDateTime	Converts a string to a *TDateTime*.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.
StrToTime	Converts a string to a time format (*TDateTime*).
StrUpper	Returns a string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type *TDateTime* to a string.
Trunc	Truncates a real number to an integer.

Table 8.3 Other standard routines (continued)

Procedure or function	Description
UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a variant array.
VarArrayDimCount	Returns number of dimensions of a variant array.
VarARrayHighBound	Returns high bound for a dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a variant array.
VarArrayOf	Creates and fills a one-dimensional variant array.
VarArrayRedim	Resizes a variant array.
VarArrayRef	Returns a reference to the passed variant array.
VarArrayUnlock	Unlocks a variant array.
VarAsType	Converts a variant to specified type.
VarCast	Converts a variant to a specified type, storing the result in a variable.
VarClear	Clears a variant.
VarCopy	Copies a variant.
VarToStr	Converts variant to string.
VarType	Returns type code of specified variant.

For information on format strings, see “Format strings” in the online Help.

Part

II

Special topics

The chapters in Part II cover specialized language features and advanced topics. These chapters include:

- Chapter 9, “Libraries and packages”
- Chapter 10, “Object interfaces”
- Chapter 11, “Memory management”
- Chapter 12, “Program control”
- Chapter 13, “Inline assembler code”

Chapter

9

Libraries and packages

A dynamically loadable library is a dynamic-link library (DLL) on Windows or a shared object library file on Linux. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

To distinguish them from standalone executables, on Windows files containing compiled DLLs are named with the .DLL extension. On Linux, files containing shared object files are named with a .so extension. Object Pascal programs can call DLLs or shared objects written in other languages, and applications written in other languages can call DLLs or shared objects written in Object Pascal.

Calling dynamically loadable libraries

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in a shared object, you must *import* them. This can be done in two ways: by declaring an **external** procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

Object Pascal does not support importing of variables from shared libraries.

Static loading

The simplest way to import a procedure or function is to declare it using the **external** directive. For example,

```
On Windows: procedure DoSomething; external 'MYLIB.DLL';

On Linux:   procedure DoSomething; external 'mylib.so';
```

If you include this declaration in a program, MYLIB.DLL (Windows) or mylib.so (Linux) is loaded once, when the program starts. Throughout execution of the program, the identifier *DoSomething* always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect **external** declarations into a separate "import unit" that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

For more information about **external** declarations, see "External declarations" on page 6-6.

Dynamic loading

You can access routines in a library through direct calls to OS library functions, including *LoadLibrary*, *FreeLibrary*, and *GetProcAddress.* In Windows, these functions are declared in Windows.pas; on Linux, they are implemented for compatibility in SysUtils.pas; the actual Linux OS routines are *dlopen*, *dlclose*, and *dlsym* (all declared in Kylix's *Libc* unit; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example, on Windows or Linux:

```
uses Windows, ...; {On Linux, replace Windows with SysUtils }

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ⋮
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

When you import routines this way, the library is not loaded until the code containing the call to *LoadLibrary* executes. The library is later unloaded by the call to *FreeLibrary*. This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.

This same example can also be written on Linux as follows:

```
uses Libc, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Pointer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ⋮
begin
  Handle := dlopen('datetime.so', RTLD_LAZY);
  if Handle <> 0 then
  begin
    @GetTime := dlsym(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    dlclose(Handle);
  end;
end;
```

In this case, when importing routines, the shared object is not loaded until the code containing the call to *dlopen* executes. The shared object is later unloaded by the call to *dlclose*. This also allows you to conserve memory and to run your program even when some of the shared objects it uses are not present.

Writing dynamically loadable libraries

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word **library** (instead of **program**).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, *Min* and *Max*.

```
library MinMax;

function Min(X, Y: Integer): Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min,
  Max;

begin
end.
```

If you want your library to be available to applications written in other languages, it's safest to specify **stdcall** in the declarations of exported functions. Other languages may not support Object Pascal's default **register** calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a **uses** clause, an **exports** clause, and the initialization code. For example,

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors,
  DoneEditors name Done,
  InsertText name Insert,
  DeleteSelection name Delete,
  FormatSelection,
  PrintSelection name Print,
   ⋮
  SetErrorHandler;

begin
  InitLibrary;
end.
```

You can put **exports** clauses in the **interface** or **implementation** section of a unit. Any library that includes such a unit in its **uses** clause automatically exports the routines listed the unit's **exports** clauses—without the need for an **exports** clause of its own.

The directive **local**, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the **local** directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for standalone procedures and functions, but not for methods. A routine declared with **local**—for example,

```
function Contraband(I: Integer): Integer; local;
```

—does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the **interface** section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the *caller* sets up EBX.

The exports clause

A routine is exported when it is listed in an **exports** clause, which has the form

exports *entry*$_1$, ..., *entry*$_n$;

where each *entry* consists of the name of a procedure, function, or variable (which must be declared prior to the **exports** clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional **name** specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive **resident**, which is maintained for backward compatibility and is ignored by the compiler.)

On Windows only, an **index** specifier consists of the directive **index** followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no **index** specifier, the routine is automatically assigned a number in the export table.

Note Use of **index** specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A **name** specifier consists of the directive **name** followed by a string constant. If an entry has no **name** specifier, the routine is exported under its original declared name, with the same spelling and case. Use a **name** clause when you want to export a routine under a different name. For example,

```
exports
  DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the **exports** clause. For example,

```
exports
  Divide(X, Y: Integer) name 'Divide_Ints',
  Divide(X, Y: Real) name 'Divide_Reals';
```

On Windows, do not include **index** specifiers in entries for overloaded routines.

An **exports** clause can appear anywhere and any number of times in the declaration part of a program or library, or in the **interface** or **implementation** section of a unit. Programs seldom contain an **exports** clause.

Library initialization code

The statements in a library's block constitute the library's *initialization code*. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an exit procedure using the *ExitProc* variable, as described in "Exit procedures" on page 12-4; the exit procedure executes when the library is unloaded.

Library initialization code can signal an error by setting the *ExitCode* variable to a nonzero value. *ExitCode* is declared in the *System* unit and defaults to zero, indicating successful initialization. If a library's initialization code sets *ExitCode* to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an exit procedure.

```
library Test;

var
  SaveExit: Pointer;

procedure LibExit;
begin
  ⋮  // library exit code
  ExitProc := SaveExit;  // restore exit procedure chain
end;

begin
  ⋮  // library initialization code
  SaveExit := ExitProc;  // save exit procedure chain
  ExitProc := @LibExit;  // install LibExit exit procedure
end.
```

When a library is unloaded, it's exit procedures are executed by repeated calls to the address stored in *ExitProc*, until *ExitProc* becomes **nil**. The initialization parts of all units used by a library are executed before the library's initialization code, and the finalization parts of those units are executed after the library's exit procedure.

Global variables in a library

Global variables declared in a shared library cannot be imported by an Object Pascal application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries—or multiple instances of a library—to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

Libraries and system variables

Several variables declared in the *System* unit are of special interest to those programming libraries. Use *IsLibrary* to determine whether code is executing in an application or in a library; *IsLibrary* is always *False* in an application and *True* in a library. During a library's lifetime, *HInstance* contains its instance handle. *CmdLine* is always **nil** in a library.

The *DLLProc* variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. *DLLProc* is available on both Windows and Linux but its use differs on each. On Windows, *DLLProc* is used in multithreading applications; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior. (See “The finalization section” on page 3-5.)

To monitor operating-system calls, create a callback procedure that takes a single integer parameter—for example,

```
procedure DLLHandler(Reason: Integer);
```

—and assign the address of the procedure to the *DLLProc* variable. When the procedure is called, it passes to it one of the following values.

DLL_PROCESS_DETACH	Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to *FreeLibrary* or (*dlclose* on Linux).
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread (Windows only).
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly (Windows only).

On Linux, these are defined in the *Libc* unit.

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

Exceptions and runtime errors in libraries

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Object Pascal, the exception can be handled through a normal **try...except** statement.

Note Under Linux this is only possible if the library and application have both been built with the same set of (base) runtime packages (which contains the EH code) or if both link to ShareExcept.

If the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code $0EEDFACE. The first entry in the *ExceptionInformation* array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Object Pascal exception object.

Generally, you should not let exceptions escape from your library. On Windows, Delphi exceptions map to the OS exception model; Linux does not have an exception model.

If a library does not use the *SysUtils* unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates.

Because the library has no way of knowing whether it was called from an Object Pascal program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

Shared-memory manager (Windows only)

On Windows, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the *ShareMem* unit. The same is true if one application or DLL allocates memory with *New* or *GetMem* which is deallocated by a call to *Dispose* or *FreeMem* in another module. *ShareMem* should always be the first unit listed in any program or library **uses** clause where it occurs.

ShareMem is the interface unit for the BORLANDMM.DLL memory manager, which allows modules to share dynamically allocated memory. BORLANDMM.DLL must be deployed with applications and DLLs that use *ShareMem*. When an application or DLL uses *ShareMem*, its memory manager is replaced by the memory manager in BORLANDMM.DLL.

Linux uses glibc's *malloc* to manage shared memory.

Packages

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange when code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

Runtime packages provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their **requires** clauses.

To distinguish them from other libraries, packages are stored in files:

- On Windows, package files end with the .bpl (Borland package library) extension.
- On Linux, packages generally begin with the prefix bpl and have a .so extension.

Ordinarily, packages are loaded statically when an applications starts. But you can use the *LoadPackage* and *UnloadPackage* routines (in the *SysUtils* unit) to load packages dynamically.

Note When an application utilizes packages, the name of each packaged unit still must appear in the **uses** clause of any source file that references it. For more information about packages, see the online Help.

Package declarations and source files

Each package is declared in a separate source file, which should be saved with the .dpk extension to avoid confusion with other files containing Object Pascal code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains

- A *name* for the package.
- A list of other packages *required* by the new package. These are packages to which the new package is linked.
- A list of unit files *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

```
package packageName;
  requiresClause;
  containsClause;
end.
```

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the *DATAX* package.

```
package DATAX;
requires
  baseclx,
  visualclx;
  contains Db, DBLocal, DBXpress, ... ;
end.
```

The **requires** clause lists other, external packages used by the package being declared. It consists of the directive **requires**, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a **requires** clause.

The **contains** clause identifies the unit files to be compiled and bound into the package. It consists of the directive **contains**, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas'; // Windows

contains MyUnit in '\home\developer\MyProject\MyUnit.pas'; // Linux
```

Note Thread-local variables (declared with **threadvar**) in a packaged unit cannot be accessed from clients that use the package.

Naming packages

A compiled package involves several generated files. For example, the source file for the package called *DATAX* is DATAX.dpk, from which the compiler generates an executable and a binary image called

- On Windows: DATAX.bpl and DATAX.dcp
- On Linux: bplDATAX.so and DATAX.dcp.

DATAX is used to refer to the package in the **requires** clauses of other packages, or when using the package in an application. Package names must be unique within a project.

The requires clause

The **requires** clause lists other, external packages that are used by the current package. It functions like the **uses** clause in a unit file. An external package listed in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's **requires** clause. If the other packages are omitted from the **requires** clause, the compiler loads the referenced units from their .dcu (Windows) or .dpu (Linux) files.

Avoiding circular package references

Packages cannot contain circular references in their **requires** clauses. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package *A* requires package *B*, then package *B* cannot require package *A*; if package *A* requires package *B* and package *B* requires package *C*, then package *C* cannot require package *A*.

Duplicate package references

The compiler ignores duplicate references in a package's **requires** clause. For programming clarity and readability, however, duplicate references should be removed.

The contains clause

The **contains** clause identifies the unit files to be bound into the package. Do not include file-name extensions in the **contains** clause.

Avoiding redundant source code uses

A package cannot be listed in the **contains** clause of another package or the **uses** clause of a unit.

All units included directly in a package's **contains** clause, or indirectly in the **uses** clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in **requires** clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

Compiling packages

Packages are ordinarily compiled from the IDE using .dpk files generated by the Package editor. You can also compile .dpk files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

Generated files

The following table lists the files produced by the successful compilation of a package.

Table 9.1 Compiled package files

File extension	Contents
dcp	A binary image containing a package header and the concatenation of all dcu (Windows) or dpu (Linux) files in the package. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file.
dcu (Windows) dpu (Linux)	A binary image for a unit file contained in a package. One dcu or dpu file is created, when necessary, for each unit file.
.bpl on Windows bpl<package>.so on Linux	The runtime package. This file is a shared library with special Borland-specific features. The base name for the package is the base name of the dpk source file.

Several compiler directives and command-line switches are available to support package compilation.

Package-specific compiler directives

The following table lists package-specific compiler directives that can be inserted into source code. See the online Help for details.

Table 9.2 Package-specific compiler directives

Directive	Purpose
{$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{$G–} or {$IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit “weakly”, as explained in the online Help.

Table 9.2 Package-specific compiler directives (continued)

Directive	Purpose
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

Including **{$DENYPACKAGEUNIT ON}** in source code prevents the unit file from being packaged. Including **{$G–}** or **{$IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages.

Other compiler directives may be included, if appropriate, in package source code.

Package-specific command-line compiler switches

The following package-specific switches are available for the command-line compiler. See the online Help for details.

Table 9.3 Package-specific command-line compiler switches

Switch	Purpose
–$G–	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
–LE *path*	Specifies the directory where the compiled package file will be placed.
–LN *path*	Specifies the directory where the package dcp file will be placed.
–LU*packageName [;packageName2;...]*	Specifies additional runtime packages to use in an application. Used when compiling a project.
–Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the **–$G–** switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

Chapter

10

Object interfaces

An *object interface*—or simply *interface*—defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models. Custom objects built that support interfaces can interact with objects written in C++, Java, and other languages.

Interface types

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
  ['{GUID}']
  memberList
end;
```

where (*ancestorInterface*) and ['{*GUID*}'] are optional. In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property **read** and **write** specifiers must be methods.

- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as **default**.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as **virtual**, **dynamic**, **abstract**, or **override**. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

In some interface declarations, the **interface** reserved word is replaced by **dispinterface**. This construction (along with the **dispid**, **readonly**, and **writeonly** directives) is platform-specific and is not used in Linux programming.

IInterface and inheritance

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not *implement* methods. What an interface inherits is the *obligation* to implement methods—an obligation that devolves onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of *IInterface*, which is defined in the *System* unit and is the ultimate ancestor of all other interfaces. *IInterface* declares three methods: *QueryInterface*, *_AddRef*, and *_Release*.

Note *IInterface* is equivalent to *IUnknown*. You should generally use *IInterface* for platform independent applications and reserve the use of *IUnknown* for specific programs that include Windows dependencies.

QueryInterface provides the means to move freely among the different interfaces that an object supports. *_AddRef* and *_Release* provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the *System* unit's *TInterfacedObject*. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects (Windows only), however, must be managed through *_AddRef* and *_Release*.

Interface identification

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

where each *x* is a hexadecimal digit (0 through 9 or A through F). On Windows, the Type Library editor automatically generates GUIDs for new interfaces; you can also generate GUIDs by pressing *Ctrl+Shift+G* in the Code editor (on Linux, you must use *Ctrl+Shift+G*).

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations. (See "Interface querying" on page 10-10.)

The *TGUID* and *PGUID* types, declared in the *System* unit, are used to manipulate GUIDs.

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

When you declare a typed constant of type *TGUID*, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type *TGUID*. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports can be called in either of two ways:

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

Calling conventions for interfaces

The default calling convention is **register**, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with **stdcall**. Use **safecall** to implement CORBA interfaces. On Windows, you can use **safecall** to implement methods of dual interfaces (as described in "Dual interfaces (Windows only)" on page 10-13).

For more information about calling conventions, see "Calling conventions" on page 6-4.

Interface properties

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled. For example, on Windows, COM objects do not have properties.

In an interface, property **read** and **write** specifiers must be methods, since fields are not available.

Forward declarations

An interface declaration that ends with the reserved word **interface** and a semicolon, without specifying an ancestor, GUID, or member list, is a *forward declaration*. A forward declaration must be resolved by a *defining declaration* of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    ⋮
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    ⋮
  end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive *IWindow* from *IControl* and also derive *IControl* from *IWindow*.

Implementing interfaces

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor. Such declarations have the form

```
type className = class (ancestorClass, interface1, ..., interfacen)
  memberList
 end;
```

For example,

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ⋮
  end;
```

declares a class called *TMemoryManager* that implements the *IMalloc* and *IErrorInfo* interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the declaration of *TInterfacedObject* in the *System* unit.

```
type
TInterfacedObject = class(TObject, IInterface)
protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

TInterfacedObject implements the *IInterface* interface. Hence *TInterfacedObject* declares and implements each of *IInterface*'s three methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares *TMemoryManager* as a direct descendent of *TInterfacedObject*.) Since every interface inherits from *IInterface*, a class that implements interfaces must implement the *QueryInterface*, *_AddRef*, and *_Release* methods. The *System* unit's *TInterfacedObject* implements these methods and is thus a convenient base from which to derive other classes that implement interfaces.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

Method resolution clauses

You can override the default name-based mappings by including *method resolution clauses* in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

```
procedure interface.interfaceMethod = implementingMethod;
```

or

```
function interface.interfaceMethod = implementingMethod;
```

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ⋮
  end;
```

maps *IMalloc*'s *Alloc* and *Free* methods onto *TMemoryManager*'s *Allocate* and *Deallocate* methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

Changing inherited implementations

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class's declaration. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ⋮
  end;

  TWindow = class(TInterfacedObject, IWindow)  // TWindow implements IWindow
    procedure Draw;
    ⋮
  end;

  TFrameWindow = class(TWindow, IWindow)  // TFrameWindow reimplements IWindow
    procedure Draw;
    ⋮
  end;
```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

Implementing interfaces by delegation

The **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

declares a property called *MyInterface* that implements the interface *IMyInterface*.

The **implements** directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a **read** specifier. If the property uses a **read** method, that method must use the default **register** calling convention and cannot be dynamic (though it can be virtual) or specify the **message** directive.

Note The class you use to implement the delegated interface should derive from *TAggregatedObject*.

Delegating to an interface-type property

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the **implements** directive, and which does so without method resolution clauses. For example,

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;

var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...  // some object whose class implements IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

Delegating to a class-type property

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
    end;
    TMyImplClass = class
      procedure P1;
      procedure P2;
    end;
    TMyClass = class(TInterfacedObject, IMyInterface)
      FMyImplClass: TMyImplClass;
      property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
      procedure IMyInterface.P1 = MyP1;
      procedure MyP1;
    end;
  procedure TMyImplClass.P1;
    ⋮
  procedure TMyImplClass.P2;
    ⋮
  procedure TMyClass.MyP1;
    ⋮
  var
    MyClass: TMyClass;
    MyInterface: IMyInterface;
  begin
    MyClass := TMyClass.Create;
    MyClass.FMyImplClass := TMyImplClass.Create;
    MyInterface := MyClass;
    MyInterface.P1;       // calls TMyClass.MyP1;
    MyInterface.P2;       // calls TImplClass.P2;
  end;
```

Interface references

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. Such variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following limitations.

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```
  type
    IAncestor = interface
    end;

    IDescendant = interface(IAncestor)
      procedure P1;
    end;

    TSomething = class(TInterfacedObject, IDescendant)
      procedure P1;
    procedure P2;
  end;
  ⋮
var
  D: IDescendant;
  A: IAncestor;
begin
  D := TSomething.Create;  // works!
  A := TSomething.Create;  // error
  D.P1;  // works!
  D.P2;  // error
end;
```

In this example,

- *A* is declared as a variable of type *IAncestor*. Because *TSomething* does not list *IAncestor* among the interfaces it implements, a *TSomething* instance cannot be assigned to *A*. But if we changed *TSomething*'s declaration to

  ```
  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
   ⋮
  ```

  the first error would become a valid assignment.
- *D* is declared as a variable of type *IDescendant*. While *D* references an instance of *TSomething*, we cannot use it to access *TSomething*'s *P2* method, since *P2* is not a method of *IDescendant*. But if we changed *D*'s declaration to

  ```
  D: TSomething;
  ```

  the second error would become a valid method call.

Interface references are managed through reference-counting, which depends on the *_AddRef* and *_Release* methods inherited from *IInterface*. When an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope.

Global interface-type variables can be initialized only to **nil**.

To determine whether an interface-type expression references an object, pass it to the standard function *Assigned*.

Interface assignment-compatibility

A class type is assignment-compatible with any interface type implemented by the class. An interface type is assignment-compatible with any ancestor interface type. The value **nil** can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type *IDispatch* or a descendant, the variant receives the type code *varDispatch*. Otherwise, the variant receives the type code *varUnknown*.

A variant whose type code is *varEmpty*, *varUnknown*, or *varDispatch* can be assigned to an *IInterface* variable. A variant whose type code is *varEmpty* or *varDispatch* can be assigned to an *IDispatch* variable.

Interface typecasts

Interface types follow the same rules as class types in variable and value typecasts. Class-type expressions can be cast to interface types—for example, `IMyInterface(SomeObject)`—provided the class implements the interface.

An interface-type expression can be cast to *Variant*. If the interface is of type *IDispatch* or a descendant, the resulting variant has the type code *varDispatch*. Otherwise, the resulting variant has the type code *varUnknown*.

A variant whose type code is *varEmpty*, *varUnknown*, or *varDispatch* can be cast to *IInterface*. A variant whose type code is *varEmpty* or *varDispatch* can be cast to *IDispatch*.

Interface querying

You can use the **as** operator to perform checked interface typecasts. This is known as *interface querying*, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

object `as` *interface*

where *object* is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and *interface* is any interface declared with a GUID.

An interface query returns **nil** if *object* is **nil**. Otherwise, it passes the GUID of *interface* to the *QueryInterface* method in *object*, raising an exception unless *QueryInterface* returns zero. If *QueryInterface* returns zero (indicating that *object*'s class implements *interface*), the interface query returns an interface reference to *object*.

Automation objects (Windows only)

An object whose class implements the *IDispatch* interface (declared in the *System* unit) is an Automation object. Automation is available on Windows only.

Dispatch interface types (Windows only)

Dispatch interface types define the methods and properties that an Automation object implements through *IDispatch*. Calls to methods of a dispatch interface are routed through *IDispatch*'s *Invoke* method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

where `['{`*GUID*`}']` is optional and *memberList* consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
  end;
```

Dispatch interface methods (Windows only)

Methods of a dispatch interface are prototypes for calls to the *Invoke* method of the underlying *IDispatch* implementation. To specify an Automation dispatch ID for a method, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.

A method declared in a dispatch interface cannot contain directives other than **dispid**. Parameter and result types must be automatable—that is, they must be *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool*, or any interface type.

Dispatch interface properties

Properties of a dispatch interface do not include access specifiers. They can be declared as **readonly** or **writeonly**. To specify a dispatch ID for a property, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as **default**. No other directives are allowed in dispatch-interface property declarations.

Accessing Automation objects (Windows only)

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include *ComObj* in the **uses** clause of one of your units or your program or library.

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The *CreateOleObject* function (defined in *ComObj*) returns an *IDispatch* reference to an Automation object and is assignment-compatible with the variant *Word*.

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of *varByte* are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the *VarArrayLock* and *VarArrayUnlock* routines.

Automation object method-call syntax

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both *positional* and *named* parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the **:=** symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type *Byte*, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *AnsiString*, *WordBool*, or *Variant*. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

Dual interfaces (Windows only)

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from *IDispatch*.

All methods of a dual interface (except from those inherited from *IInterface* and *IDispatch*) must use the **safecall** convention, and all method parameter and result types must be automatable. (The automatable types are *Byte*, *Currency*, *Real*, *Double*, *Real48*, *Integer*, *Single*, *Smallint*, *AnsiString*, *ShortString*, *TDateTime*, *Variant*, *OleVariant*, and *WordBool*.)

Chapter

11

Memory management

This chapter explains how programs use memory and describes the internal formats of Object Pascal data types.

The memory manager (Windows only)

Note Linux uses glibc functions such as *malloc* for memory management. For information, refer to the *malloc* man page on your Linux system.

On Windows systems, the memory manager manages all dynamic memory allocations and deallocations in an application. The *New*, *Dispose*, *GetMem*, *ReallocMem*, and *FreeMem* standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

On Windows, the memory manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of *GlobalAlloc*, *LocalAlloc*, and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the memory manager interfaces directly with the Win32 virtual memory API (the *VirtualAlloc* and *VirtualFree* functions). The memory manager reserves memory from the operating system in 1-MB sections of address space, and commits memory as required in 16-KB increments. It decommits and releases unused memory in 16-KB and 1-MB sections. For smaller blocks, committed memory is further suballocated.

Memory manager blocks are always rounded upward to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.

The memory manager maintains two status variables, *AllocMemCount* and *AllocMemSize*, which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging.

The *System* unit provides two procedures, *GetMemoryManager* and *SetMemoryManager*, that allow applications to intercept low-level memory manager calls. The *System* unit also provides a function called *GetHeapStatus* that returns a record containing detailed memory-manager status information. For further information about these routines, see the online Help.

Variables

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

Note On Linux, stack size is set by the environment only.

On Windows, an application's stack is defined by two values: the *minimum stack size* and the *maximum stack size*. The values are controlled through the **$MINSTACKSIZE** and **$MAXSTACKSIZE** compiler directives, and default to 16,384 (16K) and 1,048,576 (1M) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Windows application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an *EStackOverflow* exception is raised. (Stack overflow checking is completely automatic. The **$S** compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

On Windows or Linux, dynamic variables created with the *GetMem* or *New* procedure are heap-allocated and persist until they are deallocated with *FreeMem* or *Dispose*.

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

Internal data formats

The following sections describe the internal formats of Object Pascal data types.

Integer types

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range –128..127 (*Shortint*), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (*Byte*), the variable is stored as an unsigned byte.
- If both bounds are within the range –32768..32767 (*Smallint*), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (*Word*), the variable is stored as an unsigned word.
- If both bounds are within the range –2147483648..2147483647 (*Longint*), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (*Longword*), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (*Int64*).

Character types

A *Char*, an *AnsiChar*, or a subrange of a *Char* type is stored as an unsigned byte. A *WideChar* is stored as an unsigned word.

Boolean types

A *Boolean* type is stored as a *Byte*, a *ByteBool* is stored as a *Byte*, a *WordBool* type is stored as a *Word*, and a *LongBool* is stored as a *Longint*.

A *Boolean* can assume the values 0 (*False*) and 1 (*True*). *ByteBool*, *WordBool*, and *LongBool* types can assume the values 0 (*False*) or nonzero (*True*).

Enumerated types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the **{$Z1}** state (the default). If an enumerated type has more than 256 values, or if the type was declared in the **{$Z2}** state, it is stored as an unsigned word. If an enumerated type is declared in the **{$Z4}** state, it is stored as an unsigned double-word.

Real types

The real types store the binary representation of a sign (+ or –), an *exponent*, and a *significand*. A real value has the form

$$+/- \textit{significand} * 2^{\textit{exponent}}$$

where the *significand* has a single bit to the left of the binary decimal point. (That is, 0 <= *significand* < 2.)

In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses. For example, for a *Real48* value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real48 type

A 6-byte (48-bit) *Real48* number is divided into three fields:

1	39	8
s	*f*	*e*

If 0 < *e* <= 255, the value *v* of the number is given by

$$v = (-1)^{s} * 2^{(e-129)} * (1.f)$$

If *e* = 0, then *v* = 0.

The *Real48* type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a *Real48*, while NaNs and infinities produce an overflow error if an attempt is made to store them in a *Real48*.

The Single type

A 4-byte (32-bit) *Single* number is divided into three fields:

1	8	23
s	*e*	*f*

The value *v* of the number is given by

if $0 < e < 255$, then $v = (-1)^{s} * 2^{(e-127)} * (1.f)$

if $e = 0$ and $f <> 0$, then $v = (-1)^{s} * 2^{(-126)} * (0.f)$
if $e = 0$ and $f = 0$, then $v = (-1)^{s} * 0$
if $e = 255$ and $f = 0$, then $v = (-1)^{s} * \text{Inf}$
if $e = 255$ and $f <> 0$, then v is a NaN

The Double type

An 8-byte (64-bit) *Double* number is divided into three fields:

1	11	52
s	*e*	*f*

The value *v* of the number is given by

if $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$

if $e = 0$ and $f <> 0$, then $v = (-1)^s * 2^{(-1022)} * (0.f)$
if $e = 0$ and $f = 0$, then $v = (-1)^s * 0$
if $e = 2047$ and $f = 0$, then $v = (-1)^s *$ Inf
if $e = 2047$ and $f <> 0$, then v is a NaN

The Extended type

A 10-byte (80-bit) *Extended* number is divided into four fields:

1	15	1	63
s	*e*	i	*f*

The value *v* of the number is given by

if $0 <= e < 32767$, then $v = (-1)^s * 2^{(e-16383)} * (i.f)$

if $e = 32767$ and $f = 0$, then $v = (-1)^s *$ Inf
if $e = 32767$ and $f <> 0$, then v is a NaN

The Comp type

An 8-byte (64-bit) *Comp* number is stored as a signed 64-bit integer.

The Currency type

An 8-byte (64-bit) *Currency* number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Pointer types

A *Pointer* type is stored in 4 bytes as a 32-bit address. The pointer value **nil** is stored as zero.

Short string types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (**string**[255]).

Long string types

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

Table 11.1 Long string dynamic memory layout

Offset	Contents
–8	32-bit reference-count
–4	length in bytes
0..*Length* – 1	character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of –1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of –1.

Wide string types

Note On Linux, wide strings are implemented exactly as long strings.

On Windows, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide-string memory block on Windows.

Table 11.2 Wide string dynamic memory layout (Windows only)

Offset	Contents
–4	32-bit length indicator (in bytes)
0..*Length* – 1	character string
Length	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

Set types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to

(*Max* div 8) – (*Min* div 8) + 1

where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is

(*E* div 8) – (*Min* div 8)

and the bit number within that byte is

E mod 8

where *E* denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic *Integer* type or if the program contains code that takes the address of the set.

Static array types

A static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

Dynamic array types

A dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is **nil** and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

Table 11.3 Dynamic array memory layout

Offset	Contents
–8	32-bit reference-count
–4	32-bit length indicator (number of elements)
0..*Length* * (size of element) – 1	array elements

Record types

When a record type is declared in the **{$A+}** state (the default), and when the declaration does not include a **packed** modifier, the type is an *unpacked record type*, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

Table 11.4 Type alignment masks

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for *Real48*, 4 for *Single*, 8 for *Double* and *Extended*
Short string types	1
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record
Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	4

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

When a record type is declared in the **{$A–}** state, or when the declaration includes the **packed** modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

File types

File types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```
type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
```

```
            BufEnd: Cardinal;
            BufPtr: PChar;
            OpenFunc: Pointer;
            InOutFunc: Pointer;
            FlushFunc: Pointer;
            CloseFunc: Pointer;
            UserData: array[1..32] of Byte;
            Name: array[0..259] of Char; );
    end;
```

Text files occupy 460 bytes, which are laid out as follows:

```
type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
    Buffer: TTextBuf;
  end;
```

Handle contains the file's handle (when the file is open).

The *Mode* field can assume one of the values

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file; see “Device functions” on page 8-5. Flags determines the line break style as follows:

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

All other Flags bits are reserved for future use. See also *DefaultTextLineBreakStyle* and *SetLineBreakStyle*.

Procedural types

A procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

Class types

A class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object’s fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMTs, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointers—one per user-defined virtual method in the class type—in order of declaration. Each slot contains the address of the corresponding virtual method’s entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Object Pascal’s implementation. Applications should use the methods defined in *TObject* to query this information, since the layout is likely to change in future implementations of Object Pascal.

Table 11.5 Virtual method table layout

Offset	Type	Description
–76	*Pointer*	pointer to virtual method table (or **nil**)
–72	*Pointer*	pointer to interface table (or **nil**)
–68	*Pointer*	pointer to Automation information table (or **nil**)
–64	*Pointer*	pointer to instance initialization table (or **nil**)
–60	*Pointer*	pointer to type information table (or **nil**)

Table 11.5 Virtual method table layout (continued)

Offset	Type	Description
–56	*Pointer*	pointer to field definition table (or **nil**)
–52	*Pointer*	pointer to method definition table (or **nil**)
–48	*Pointer*	pointer to dynamic method table (or **nil**)
–44	*Pointer*	pointer to short string containing class name
–40	*Cardinal*	instance size in bytes
–36	*Pointer*	pointer to a pointer to ancestor class (or **nil**)
–32	*Pointer*	pointer to entry point of *SafecallException* method (or **nil**)
–28	*Pointer*	entry point of *AfterConstruction* method
–24	*Pointer*	entry point of *BeforeDestruction* method
–20	*Pointer*	entry point of *Dispatch* method
–16	*Pointer*	entry point of *DefaultHandler* method
–12	*Pointer*	entry point of *NewInstance* method
–8	*Pointer*	entry point of *FreeInstance* method
–4	*Pointer*	entry point of *Destroy* destructor
0	*Pointer*	entry point of first user-defined virtual method
4	*Pointer*	entry point of second user-defined virtual method
⋮	⋮	⋮

Class reference types

A class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

Variant types

A variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The *System* and *Variants* units define constants and types for variants.

The *TVarData* type represents the internal structure of a *Variant* variable (on Windows, this is identical to the *Variant* type used by COM and the Win32 API). The *TVarData* type can be used in typecasts of *Variant* variables to access the internal structure of a variable.

The *VType* field of a *TVarData* record contains the type code of the variant in the lower twelve bits (the bits defined by the *varTypeMask* constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.

The *Reserved1*, *Reserved2*, and *Reserved3* fields of a *TVarData* record are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field. If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.

If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines an array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.

If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and *varArray* bits in the *VType* field.

The *varString* type code is private. Variants containing a *varString* value should never be passed to a non-Delphi function. On Windows, Delphi's Automation support automatically converts *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

On Linux, *VT_decimal* is not supported.

Chapter

12

Program control

This chapter explains how parameters and function results are stored and transferred. The final section discusses exit procedures.

Parameters and function results

Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

Parameter passing

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see "Calling conventions" on page 6-4.

Variable (**var**) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (**const**) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A *Single* parameter occupies 4 bytes, and a *Double*, *Comp*, or *Currency* parameter occupies 8 bytes. A *Real48* occupies 8 bytes, with the *Real48* value stored in the lower 6 bytes. An *Extended* occupies 12 bytes, with the *Extended* value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.

- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value **nil** is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the **register** and **pascal** conventions, a variant parameter is passed as a 32-bit pointer to a *Variant* value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32-bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack under the **cdecl**, **stdcall**, and **safecall** conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.
- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

Under the **pascal**, **cdecl**, **stdcall** and **safecall** conventions, all parameters are passed on the stack. Under the **pascal** convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up at the highest address and the last parameter ends up at the lowest address. Under the **cdecl**, **stdcall**, and **safecall** conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

Under the **register** convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the **pascal** convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, *Int64*, and structured types do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to *Test* passes *A* in EAX as a 32-bit integer, *B* in EDX as a pointer to a *Char*, and *D* in ECX as a pointer to a long-string memory block; *C* and *E* are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a CLD instruction) and must return with the direction flag cleared.

Function results

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type *Currency*, the value in ST(0) is scaled by 10000. For example, the *Currency* value 1.234 is returned in ST(0) as 12340.
- For a string, dynamic array, method pointer, variant, or *Int64* result, the effects are the same as if the function result were declared as an additional **var** parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.
- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional **var** parameter that is passed to the function after the declared parameters.

Method calls

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter *Self*, which is a reference to the instance or class in which the method is called. The *Self* parameter is passed as a 32-bit pointer.

- Under the **register** convention, *Self* behaves as if it were declared *before* all other parameters. It is therefore always passed in the EAX register.
- Under the **pascal** convention, *Self* behaves as if it were declared *after* all other parameters (including the additional **var** parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.
- Under the **cdecl**, **stdcall**, and **safecall** conventions, *Self* behaves as if it were declared *before* all other parameters, but *after* the additional **var** parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional **var** parameter.

Constructors and destructors

Constructors and destructors use the same calling conventions as other methods, except that an additional *Boolean* flag parameter is passed to indicate the context of the constructor or destructor call.

A value of *False* in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the **inherited** keyword. In this case, the constructor behaves like an ordinary method. A value of *True* in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by *Self*, and returns a reference to the newly created object in EAX.

A value of *False* in the flag parameter of a destructor call indicates that the destructor was invoked using the **inherited** keyword. In this case, the destructor behaves like an ordinary method. A value of *True* in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by *Self* just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the **register** convention, it is passed in the DL register. Under the **pascal** convention, it is pushed before all other parameters. Under the **cdecl**, **stdcall**, and **safecall** conventions, it is pushed just before the *Self* parameter.

Since the DL register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of DL before exiting so that *BeforeDestruction* or *AfterConstruction* can be called properly.

Exit procedures

Exit procedures ensure that specific actions—such as updating and closing files—are carried out before a program terminates. The *ExitProc* pointer variable allows you to "install" an exit procedure, so that it is always called as part of the program's termination—whether the termination is normal, forced by a call to *Halt*, or the result of a runtime error. An exit procedure takes no parameters.

Note It is recommended that finalization sections, rather than exit procedures, be used for all exit behavior. (See "The finalization section" on page 3-5.) Exit procedures are available only for executables, shared objects (Linux) or .DLL (Windows) targets; for packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn't executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents of *ExitProc* before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of *ExitProc*.

The following code shows a skeleton implementation of an exit procedure.

```
var
  ExitSave: Pointer;

procedure MyExit;
begin
  ExitProc := ExitSave;  // always restore old vector first
  ⋮
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  ⋮
end.
```

On entry, the code saves the contents of *ExitProc* in *ExitSave*, then installs the *MyExit* procedure. When called as part of the termination process, the first thing *MyExit* does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until *ExitProc* becomes **nil**. To avoid infinite loops, *ExitProc* is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to *ExitProc*. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable. In case of normal termination, *ExitCode* is zero and *ErrorAddr* is **nil**. In case of termination through a call to *Halt*, *ExitCode* contains the value passed to *Halt* and *ErrorAddr* is **nil**. In case of termination due to a runtime error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the *Input* and *Output* files. If *ErrorAddr* is not **nil**, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines *ErrorAddr* and outputs a message if it’s not **nil**; before returning, set *ErrorAddr* to **nil** so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in *ExitCode* as a return code.

Chapter

13

Inline assembler code

The built-in assembler allows you to write assembler code within Object Pascal programs. It has the following features:

- Allows for inline assembly
- Supports all instructions found in the Intel Pentium III, SIMD, and the AMD Athlon (including 3D Now!)
- Provides no macro support, but allows for pure assembler function procedures
- Permits the use of Object Pascal identifiers, such as constants, types, and variables in assembler statements

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See "Linking to object files" on page 6-6 for more information.

Note If you have external assembler code that you want to use in your applications, you should consider rewriting it in Object Pascal or minimally reimplement it using the inline assembler.

The asm statement

The built-in assembler is accessed through **asm** statements, which have the form

`asm` *statementList* `end`

where *statementList* is a sequence of assembler statements separated by semicolons, end-of-line characters, or Object Pascal comments.

Comments in an **asm** statement must be in Object Pascal style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word **inline** and the directive **assembler** are maintained for backward compatibility only. They have no effect on the compiler.

Register use

In general, the rules of register use in an **asm** statement are the same as those of an **external** procedure or function. An **asm** statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an **asm** statement, BP points to the current stack frame, SP points to the top of the stack, SS contains the segment address of the stack segment, and DS contains the segment address of the data segment. Except for ESP and EBP, an **asm** statement can assume nothing about register contents on entry to the statement.

Assembler statement syntax

This syntax of an assembler statement is

Label: *Prefix Opcode Operand*$_1$, *Operand*$_2$

where *Label* is a label, *Prefix* is an assembler prefix opcode (operation code), *Opcode* is an assembler instruction opcode or directive, and *Operand* is an assembler expression. *Label* and *Prefix* are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembler statements, but not within them. For example,

```
MOV AX,1 {Initial value}     { OK }
MOV CX,100 {Count}           { OK }

MOV {Initial value} AX,1;    { Error! }
MOV CX, {Count} 100          { Error! }
```

Labels

Labels are used in built-in assembler statements as they are in Object Pascal—by writing the label and a colon before a statement. There is no limit to a label's length. As in Object Pascal, labels must be declared in a **label** declaration part in the block containing the **asm** statement. There is one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to **asm** statements, and the scope of a local label extends from the **asm** reserved word to the end of the **asm** statement that contains it. A local label doesn't have to be declared.

Instruction opcodes

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family

- Pentium Pro and Pentium II
- Pentium III
- Pentium IV

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

RET instruction sizing

The RET instruction opcode always generates a near return.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is –128 to 127 bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is –128 to 127 bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembler statement

```
JC       Stop
```

where *Stop* isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
JNC      Skip
JMP      Stop
Skip:
```

Jumps to the entry points of procedures and functions are always near.

Assembler directives

The built-in assembler supports three assembler define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between –128 and 255, or a character string of any length.

Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between –32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer—that is, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between –2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer—that is, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The DQ directive defines a quadword for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembler statements. To generate uninitialized or initialized data in the data segment, you should use Object Pascal **var** or **const** declarations.

Some examples of DB, DW, and DD directives follow.

```
asm
  DB      0FFH                      { One byte }
  DB      0,99                      { Two bytes }
  DB      'A'                       { Ord('A') }
  DB      'Hello world...',0DH,0AH  { String followed by CR/LF }
  DB      12,"string"               { Object Pascal style string }
  DW      0FFFFH                    { One word }
  DW      0,9999                    { Two words }
  DW      'A'                       { Same as DB 'A',0 }
  DW      'BA'                      { Same as DB 'A','B' }
  DW      MyVar                     { Offset of MyVar }
  DW      MyProc                    { Offset of MyProc }
  DD      0FFFFFFFFH                { One double-word }
  DD      0,999999999               { Two double-words }
  DD      'A'                       { Same as DB 'A',0,0,0 }
  DD      'DCBA'                    { Same as DB 'A','B','C','D' }
  DD      MyVar                     { Pointer to MyVar }
  DD      MyProc                    { Pointer to MyProc }
end;
```

In Turbo Assembler, when an identifier precedes a DB, DW, or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, Turbo Assembler allows the following:

```
ByteVar     DB     ?
WordVar     DW     ?
IntVar      DD     ?
  ⋮
            MOV    AL,ByteVar
            MOV    BX,WordVar
            MOV    ECX,IntVar
```

The built-in assembler doesn’t support such variable declarations. The only kind of symbol that can be defined in an inline assembler statement is a label. All variables must be declared using Object Pascal syntax; the preceding construction can be replaced by

```
var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
  ⋮
asm
  MOV     AL,ByteVar
  MOV     BX,WordVar
  MOV     ECX,IntVar
end;
```

SMALL and LARGE can be used determine the width of a displacement:

```
MOV eax, [large $1234]
```

This instruction generates a "normal" move with a 32-bit displacement ($00001234).

```
MOV eax, [small $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement ($1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
MOV eax, [SMALL 123]
```

as opposed to

```
mov eax, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual method: VMTOFFSET and DMTINDEX.

VMTOFFSET retrives the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter, for example,TExample.VirtualMethod.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example,TExample.DynamicMethod. To invoke the dynamic method, call System.@CallDynaInst with the (E)SI register containing the value obtained from DMTINDEX.

Note Methods with the "message" directive, are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```
TMyClass = class
  procedure x; message MYMESSAGE;
end;
```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```
program Project2;

type
  TExample = class
    procedure DynamicMethod; dynamic;
    procedure VirtualMethod; virtual;
  end;

procedure TExample.DynamicMethod;
begin

end;

procedure TExample.VirtualMethod;
begin

end;

procedure CallDynamicMethod(e: TExample);
asm
  // Save ESI register
  PUSH    ESI

  // Instance pointer needs to be in EAX
  MOV     EAX, e
  // DMT entry index needs to be in (E)SI
  MOV     ESI, DMTINDEX TExample.DynamicMethod
  // Now call the method
  CALL    System.@CallDynaInst

  // Restore ESI register
  POP ESI
end;

procedure CallVirtualMethod(e: TExample);
asm
  // Instance pointer needs to be in EAX
  MOV     EAX, e
```

```
    // Retrieve VMT table entry
    MOV     EDX, [EAX]

    // Now call the method at offset VMTOFFSET
    CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]
  end;

  var
    e: TExample;

  begin
    e := TExample.Create;
    try
      CallDynamicMethod(e);
      CallVirtualMethod(e);
    finally
      e.Free;
    end;
  end.
```

Operands

Built-in assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings

Table 13.1 Built-in assembler reserved words

AH	BYTE	DMTINDEX	EDI	HIGH	QWORD	TBYTE
AL	CH	DS	EDX	LARGE	SHL	TYPE
AND	CL	DWORD	EIP	LOW	SHR	VMTOFFSET
AX	CS	DX	ES	MOD	SI	WORD
BH	CX	EAX	ESI	NOT	SMALL	XOR
BL	DH	EBP	ESP	OFFSET	SP	
BP	DI	EBX	FS	OR	SS	
BX	DL	ECX	GS	PTR	ST	

Reserved words always take precedence over user-defined identifiers. For example,

```
var
  Ch: Char;
  ⋮
asm
  MOV     CH, 1
end;
```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) override operator:

```
MOV     &Ch, 1
```

It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

Expressions

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants.

Expressions are built from *expression elements* and *operators*, and each expression has an associated *expression class* and *expression type*.

Differences between Object Pascal and assembler expressions

The most important difference between Object Pascal expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value—a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid statement.

```
asm
  MOV     Z,X+Y
end;
```

Because both *X* and *Y* are constants, the expression `X + Y` is a convenient way of writing the constant 30, and the resulting instruction simply moves of the value 30 into the variable *Z*. But if *X* and *Y* are variables—

```
var
  X, Y: Integer;
```

—the built-in assembler cannot compute the value of `X + Y` at compile time. In this case, to move the sum of *X* and *Y* into *Z* you would use

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

In an Object Pascal expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Object Pascal the expression `X + 4` (where *X* is a variable) means the contents of X plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of *X*. So, even though you're allowed to write

```
asm
```

```
  MOV     EAX,X+4
end;
```

this code doesn’t load the value of *X* plus 4 into AX; instead, it loads the value of a word stored four bytes beyond *X*. The correct way to add 4 to the contents of *X* is

```
asm
  MOV     EAX,X
  ADD     EAX,4
end;
```

Expression elements

The elements of an expression are *constants*, *register*s, and *symbols*.

Constants

The built-in assembler supports two types of constant: *numeric constants* and *string constants*.

Numeric constants

Numeric constants must be integers, and their values must be between –2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a *B* after the number, octal notation by writing an *O* after the number, and hexadecimal notation by writing an *H* after the number or a **$** before the number.

Numeric constants must start with one of the digits 0 through 9 or the **$** character. When you write a hexadecimal constant using the *H* suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example, `0BAD4H` and `$BAD4` are hexadecimal constants, but `BAD4H` is an identifier because it starts with a letter.

String constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
‘Linux’
"That's all folks"
'"That''s all folks," he said.'
'100'
'"'
"'"
```

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

Table 13.2 String examples and their values

String	Value
`'a'`	00000061H
`'ba'`	00006261H
`'cba'`	00636261H
`'dcba'`	64636261H
`'a '`	00006120H
`'   a'`	20202061H
`'a' * 2`	000000E2H
`'a'-'A'`	00000020H
not `'a'`	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers:.

Table 13.3 CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a *register operand*. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registers—for example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(*X*), where *X* is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Object Pascal identifiers in assembler expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol *@Result*, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

could be written in assembler as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

The following symbols cannot be used in **asm** statements:

- Standard procedures and functions (for example, *WriteLn* and *Chr*).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The *@Result* symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in **asm** statements.

Table 13.4 Symbols recognized by the built-in assembler

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
@Result	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV     EAX,Count
```

assembles into MOV EAX,[EBP–4].

The built-in assembler treats **var** parameters as a 32-bit pointers, and the size of a **var** parameter is always 4. The syntax for accessing a **var** parameter is different from that for accessing a value parameter. To access the contents of a **var** parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    MOV     EAX,[EAX]
    MOV     EDX,Y
    ADD     EAX,[EDX]
    MOV     @Result,EAX
  end;
end;
```

Identifiers can be qualified within **asm** statements. For example, given the declarations

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

the following constructions can be used in an **asm** statement to access fields.

```
MOV     EAX,P.X
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of [EDX] into EAX.

```
MOV     EAX,(TRect PTR [EDX]).B.X
MOV     EAX,TRect([EDX]).B.X
MOV     EAX,TRect[EDX].B.X
MOV     EAX,[EDX].TRect.B.X
```

Expression classes

The built-in assembler divides expressions into three classes: *registers*, *memory references*, and *immediate values*.

An expression that consists solely of a register name is a *register* expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Object Pascal's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Object Pascal's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;
  ⋮
asm
  MOV     EAX,Start            { MOV EAX,xxxx }
  MOV     EBX,Count            { MOV EBX,[xxxx] }
  MOV     ECX,[Start]          { MOV ECX,[xxxx] }
  MOV     EDX,OFFSET Count     { MOV EDX,xxxx }
end;
```

Because *Start* is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as *Count* is a memory reference. In the third MOV, the brackets convert *Start* into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth MOV, the OFFSET operator converts *Count* into an immediate value (the offset of *Count* in the data segment).

The brackets and OFFSET operator complement each other. The following **asm** statement produces identical machine code to the first two lines of the previous **asm** statement.

```
asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either *relocatable* or *absolute*. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression types

Every built-in assembler expression has a type—or, more correctly, a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an *Integer* variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  ⋮
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

the assembler checks that the size of *QuitFlag* is one (a byte), and that the size of *OutBufPtr* is two (a word). The instruction

```
MOV     DL,OutBufPtr
```

produces an error because DL is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

These MOV instructions all refer to the first (least significant) byte of the *OutBufPtr* variable.

In some cases, a memory reference is untyped. One example is an immediate value enclosed in square brackets:

```
MOV al, [Buffer]
MOV cx, [Buffer]
MOV edx, [Buffer]
```

The built-in assembler permits both of these instructions, because the expression [100H] has no type—it just means "the contents of address 100H in the data segment," and the type can be determined from the first operand (byte for AL, word for BX). In cases where the type can't be determined from another operand, the built-in assembler requires an explicit typecast:

```
INC     BYTE PTR [ECX]
IMUL    WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Object Pascal types.

Table 13.5 Predefined type symbols

Symbol	Type
BYTE	1
WORD	2

Table 13.5 Predefined type symbols (continued)

Symbol	Type
DWORD	4
QWORD	8
TBYTE	10

Expression operators

The built-in assembler provides a variety of operators. Precedence rules are different from Object Pascal; for example, in an **asm** statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

Table 13.6 Precedence of built-in assembler expression operators

Operators	Remarks	Precedence
&		highest
(), **[]**, **.**, **HIGH**, **LOW**		
+, **–**	unary **+** and **–**	
:		
OFFSET, **TYPE**, **PTR**, *****, **/**, **MOD**, **SHL**, **SHR**, **+**, **–**	binary **+** and **–**	
NOT, **AND**, **OR**, **XOR**		lowest

The following table defines the built-in assembler's expression operators.

Table 13.7 Definitions of built-in assembler expression operators

Operator	Description
&	**Identifier override**. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	**Subexpression**. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference**. The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector**. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.

Table 13.7 Definitions of built-in assembler expression operators (continued)

Operator	Description
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus**. Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
–	**Unary minus**. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	**Addition**. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
–	**Subtraction**. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	**Segment override**. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator**. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	**Multiplication**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembler procedures and functions

You can write complete procedures and functions using inline assembler code, without including a **begin...end** statement. For example,

```
function LongMul(X, Y: Integer): Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were **var** parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a reference to the *@Result* symbol is an error. For strings, variants, and interfaces, the caller always allocates an *@Result* pointer.
- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- The automatically generated entry and exit code for the routine looks like this:

```
PUSH    EBP             ;Present if Locals <> 0 or Params <> 0
MOV     EBP,ESP         ;Present if Locals <> 0 or Params <> 0
SUB     ESP,Locals      ;Present if Locals <> 0
 ⋮
MOV     ESP,EBP         ;Present if Locals <> 0
POP     EBP             ;Present if Locals <> 0 or Params <> 0
RET     Params          ;Always present
```

  If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.

- *Locals* is the size of the local variables and *Params* is the size of the parameters. If both *Locals* and *Params* are zero, there is no entry code, and the exit code consists simply of a RET instruction.

Assembler functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (*Currency* values are scaled by 10000.)
- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by *@Result*.

Appendix A

Object Pascal grammar

```
Goal -> (Program | Package  | Library  | Unit)

Program -> [PROGRAM Ident ['(' IdentList ')'] ';']
           ProgramBlock '.'

Unit -> UNIT Ident ';'
        InterfaceSection
        ImplementationSection
        InitSection '.'

Package -> PACKAGE Ident ';'
           [RequiresClause]
           [ContainsClause]
           END '.'

Library -> LIBRARY Ident ';'
           ProgramBlock '.'

ProgramBlock -> [UsesClause]
                Block

UsesClause -> USES IdentList ';'

InterfaceSection -> INTERFACE
                    [UsesClause]
                    [InterfaceDecl]...

InterfaceDecl -> ConstSection
              -> TypeSection
              -> VarSection
              -> ExportedHeading

ExportedHeading -> ProcedureHeading ';' [Directive]
                -> FunctionHeading ';' [Directive]

ImplementationSection -> IMPLEMENTATION
                         [UsesClause]
                         [DeclSection]...

Block -> [DeclSection]
```

```
         CompoundStmt

DeclSection -> LabelDeclSection
            -> ConstSection
            -> TypeSection
            -> VarSection
            -> ProcedureDeclSection

LabelDeclSection -> LABEL LabelId

ConstSection -> CONST (ConstantDecl ';')...

ConstantDecl -> Ident '=' ConstExpr
             -> Ident ':' TypeId '=' TypedConstant

TypeSection -> TYPE (TypeDecl ';')...

TypeDecl -> Ident '=' Type
         -> Ident '=' RestrictedType

TypedConstant -> (ConstExpr | ArrayConstant | RecordConstant)

ArrayConstant -> '(' TypedConstant/','... ')'

RecordConstant -> '(' RecordFieldConstant/';'... ')'

RecordFieldConstant -> Ident ':' TypedConstant

Type -> TypeId
     -> SimpleType
     -> StrucType
     -> PointerType
     -> StringType
     -> ProcedureType
     -> VariantType
     -> ClassRefType

RestrictedType -> ObjectType
               -> ClassType
               -> InterfaceType

ClassRefType -> CLASS OF TypeId

SimpleType -> (OrdinalType | RealType)

RealType -> REAL48
         -> REAL
         -> SINGLE
         -> DOUBLE
         -> EXTENDED
         -> CURRENCY
         -> COMP

OrdinalType -> (SubrangeType | EnumeratedType | OrdIdent)

OrdIdent -> SHORTINT
         -> SMALLINT
         -> INTEGER
         -> BYTE
         -> LONGINT
         -> INT64
         -> WORD
         -> BOOLEAN
         -> CHAR
         -> WIDECHAR
         -> LONGWORD
         -> PCHAR

VariantType -> VARIANT
            -> OLEVARIANT

SubrangeType -> ConstExpr '..' ConstExpr

EnumeratedType -> '(' EnumeratedTypeElement/','... ')'

EnumeratedTypeElement -> Ident [ '=' ConstExpr ]

StringType -> STRING
           -> ANSISTRING
           -> WIDESTRING
           -> STRING '[' ConstExpr ']'

StrucType -> [PACKED] (ArrayType | SetType | FileType | RecType)

ArrayType -> ARRAY ['[' OrdinalType/','... ']'] OF Type

RecType -> RECORD [FieldList] END

FieldList ->  FieldDecl/';'... [VariantSection] [';']

FieldDecl -> IdentList ':' Type

VariantSection -> CASE [Ident ':'] TypeId OF RecVariant/';'...

RecVariant -> ConstExpr/','...  ':' '(' [FieldList] ')'

SetType -> SET OF OrdinalType

FileType -> FILE OF TypeId

PointerType -> '^' TypeId

ProcedureType -> (ProcedureHeading | FunctionHeading) [OF OBJECT]

VarSection -> VAR (VarDecl ';')...

VarDecl -> IdentList ':' Type [(ABSOLUTE (Ident | ConstExpr)) | '=' ConstExpr]

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['+' | '-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator ['(' ExprList ')']
       -> '@' Designator
       -> Number
       -> String
       -> NIL
       -> '(' Expression ')'
       -> NOT Factor
       -> SetConstructor
       -> TypeId '(' Expression ')'

RelOp -> '>'
      -> '<'
      -> '<='
      -> '>='
      -> '<>'
      -> IN
      -> IS
      -> AS

AddOp -> '+'
      -> '-'
      -> OR
      -> XOR

MulOp -> '*'
      -> '/'
      -> DIV
      -> MOD
      -> AND
      -> SHL
      -> SHR

Designator -> QualId ['.' Ident | '[' ExprList ']' | '^']...

SetConstructor -> '[' [SetElement/','...] ']'

SetElement -> Expression ['..' Expression]

ExprList -> Expression/','...

Statement -> [LabelId ':'] [SimpleStatement | StructStmt]

StmtList -> Statement/';'...

SimpleStatement -> Designator ['(' ExprList ')']
                -> Designator ':=' Expression
                -> INHERITED
                -> GOTO LabelId

StructStmt -> CompoundStmt
           -> ConditionalStmt
           -> LoopStmt
           -> WithStmt

CompoundStmt -> BEGIN StmtList END

ConditionalStmt -> IfStmt
                -> CaseStmt

IfStmt -> IF Expression THEN Statement [ELSE Statement]

CaseStmt -> CASE Expression OF CaseSelector/';'... [ELSE StmtList] [';'] END

CaseSelector -> CaseLabel/','... ':' Statement

CaseLabel -> ConstExpr ['..' ConstExpr]

LoopStmt -> RepeatStmt
         -> WhileStmt
         -> ForStmt

RepeatStmt -> REPEAT Statement UNTIL Expression

WhileStmt -> WHILE Expression DO Statement

ForStmt -> FOR QualId ':=' Expression (TO | DOWNTO) Expression DO Statement

WithStmt -> WITH IdentList DO Statement

ProcedureDeclSection -> ProcedureDecl
```

```
                     -> FunctionDecl

ProcedureDecl -> ProcedureHeading ';' [Directive]
                 Block ';'

FunctionDecl -> FunctionHeading ';' [Directive]
                Block ';'

FunctionHeading -> FUNCTION Ident [FormalParameters] ':' (SimpleType | STRING)

ProcedureHeading -> PROCEDURE Ident [FormalParameters]

FormalParameters -> '(' FormalParm/';'... ')'

FormalParm -> [VAR | CONST | OUT] Parameter

Parameter -> IdentList  [':' ([ARRAY OF] SimpleType | STRING | FILE)]
          -> Ident ':' SimpleType '=' ConstExpr

Directive -> CDECL
          -> REGISTER
          -> DYNAMIC
          -> VIRTUAL
          -> EXPORT
          -> EXTERNAL
          -> FAR
          -> FORWARD
          -> MESSAGE
          -> OVERRIDE
          -> OVERLOAD
          -> PASCAL
          -> REINTRODUCE
          -> SAFECALL
          -> STDCALL

ObjectType -> OBJECT [ObjHeritage] [ObjFieldList] [MethodList] END

ObjHeritage -> '(' QualId ')'

MethodList -> (MethodHeading [';' VIRTUAL])/';'...

MethodHeading -> ProcedureHeading
              -> FunctionHeading
              -> ConstructorHeading
              -> DestructorHeading

ConstructorHeading -> CONSTRUCTOR Ident [FormalParameters]

DestructorHeading -> DESTRUCTOR Ident [FormalParameters]

ObjFieldList -> (IdentList ':' Type)/';'...

InitSection -> INITIALIZATION StmtList [FINALIZATION StmtList] END
            -> BEGIN StmtList END
            -> END

ClassType -> CLASS [ClassHeritage]
             [ClassFieldList]
             [ClassMethodList]
             [ClassPropertyList]
             END

ClassHeritage -> '(' IdentList ')'
```

```
ClassVisibility -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]

ClassFieldList -> (ClassVisibility ObjFieldList)/';'...

ClassMethodList -> (ClassVisibility MethodList)/';'...

ClassPropertyList -> (ClassVisibility PropertyList ';')...

PropertyList -> PROPERTY  Ident [PropertyInterface]  PropertySpecifiers

PropertyInterface -> [PropertyParameterList] ':' Ident

PropertyParameterList -> '[' (IdentList ':' TypeId)/';'... ']'

PropertySpecifiers -> [INDEX ConstExpr]
                      [READ Ident]
                      [WRITE Ident]
                      [STORED (Ident | Constant)]
                      [(DEFAULT ConstExpr) | NODEFAULT]
                      [IMPLEMENTS TypeId]

InterfaceType -> INTERFACE [InterfaceHeritage]
                 [ClassMethodList]
                 [ClassPropertyList]
                 END

InterfaceHeritage -> '(' IdentList ')'

RequiresClause -> REQUIRES IdentList... ';'

ContainsClause -> CONTAINS IdentList... ';'

IdentList -> Ident/','...

QualId -> [UnitId '.'] Ident

TypeId -> [UnitId '.'] <type-identifier>

Ident -> <identifier>

ConstExpr -> <constant-expression>

UnitId -> <unit-identifier>

LabelId -> <label-identifier>

Number -> <number>

String -> <string>
```

Index

Rave Reports

Borland Edition 5.0

Developers Guide

Tutorial and Reference

Table of Contents

Technical Information

Single User License Agreement

This is a legal Agreement between you, as the end user, and Nevrona Designs. By opening the enclosed sealed disk package, or by using the disk, you are agreeing to be bound by the terms of this Agreement. If you do not agree with the terms of this Agreement, promptly return the unopened disk package and accompanying items, (including written materials), to the place you obtained them for a full refund.

1. **Grant of License** - Nevrona Designs grants to you the right to use one copy of the enclosed Nevrona Designs program, (the Software), on a single terminal connected to a single computer (i.e. CPU). You may make one copy of the Software for back-up purposes for use on your own computer. You must reproduce and include the copyright notice on the back-up copy. You may not network the Software or use it on more than a single computer or computer terminal at any time, unless a copy is purchased for each computer or terminal on the network that will use the Software. You may transfer this Software from one computer to another, provided that the Software is used on only one computer at a time. You may not rent or lease the Software, but you may transfer the Software and accompanying written material and this license to another person on a permanent basis provided you retain no copies and the other person agrees to accept the terms and conditions of this Agreement. THIS SOFTWARE MAY NOT BE DISTRIBUTED, IN MODIFIED OR UNMODIFIED FORM, AS PART OF ANY APPLICATION PROGRAM OR OTHER SOFTWARE THAT IS A LIBRARY-TYPE PRODUCT, DEVELOPMENT TOOL OR OPERATING SYSTEM, OR THAT MAY BE COMPETITIVE WITH, OR USED IN LIEU OF, THE PROGRAM PRODUCT, WITHOUT THE EXPRESS WRITTEN PERMISSION OF NEVRONA DESIGNS. This license does include the right to distribute applications using the enclosed software provided the above requirements are met.

2. **Term** - This Agreement is effective until you terminate it by destroying the Software, together with all copies. It will also terminate if you fail to follow this agreement. You agree upon termination to destroy the Software, together with all copies thereof.

3. **Copyright** - The software is owned by Nevrona Designs and is protected by United States laws and international treaty provisions. Therefore, you must treat the Software like any other copyrighted material (e.g. a book or musical recording) **EXCEPT** that you may either (a) make one copy of the Software solely for back-up or archival purposes, or (b) transfer the Software to a single hard disk provided you keep the original solely for back-up or archival purposes. You may not copy the written materials accompanying the Software.

Limited Warranty

1. **Limited Warranty -** Nevrona Designs warrants that the disks on which the Software is furnished to be free from defects in material and workmanship, under normal use, for a period of 90 days after the date of the original purchase. If, during this 90-day period, a defect in the disk should occur, the disk may be returned with proof of purchase to Nevrona Designs, which will replace the disk without charge. Nevrona Designs warrants that the Software will perform substantially in accordance with the accompanying written materials. Nevrona Designs does not warrant that the functions contained in the Software will meet your requirements, or any operation of the Software will be uninterrupted or error-free. However, Nevrona Designs will, after being notified of significant errors during the 90-day period, correct demonstrable and significant Software or documentation errors within a reasonable period of time, or refund all or a fair portion of the price you have paid for the Software at Nevrona Designs' option.

2. **Disclaimer of Warranties - Nevrona Designs disclaims all other warranties, either**

expressed or implied, including but not limited to implied warranties of merchantability of fitness from particular purpose, with respect to the Software and accompanying written materials. This limited warranty gives you specific legal rights, you may have others, varying from state to state. Nevrona Designs will have no consequential damages. In no event, shall Nevrona Designs or its suppliers be liable for damages whatsoever, (including without limitation, damages for loss of business profits, business interruption, loss of business information, or any pecuniary loss), arising out of the use or the inability to this Nevrona Designs product, even if Nevrona Designs has been advised of the possibility of such damages. Some states do not allow the exclusion of limitation of liability for consequential or incidental damages, and this limitation may not apply to you.

3. **Sole Remedy -** Nevrona Designs' entire liability in your inclusive remedy shall be, at Nevrona Designs' option, either: (1) The return of the purchase price paid; or (2) Repair or replacement of the Software that does not meet Nevrona Designs' limited warranty, which is returned to Nevrona Designs with a copy of your receipt.

4. **Governing Law** - This Agreement will be construed and governed in accordance with laws of the State of Arizona.

5. **U.S. Government Restricted Rights -** This Software and documentation are provided with restrictive rights. Use, duplication or disclosure by the Government is subject to restrictions set forth in Section c(1)(ii) of the Rights and Technical Data in Computer Software clause at 52.227-7013.

Technical Support

Technical support, product updates, addons and other information relating to Rave Reports can be found at the Nevrona Designs web site. Please visit one of the following web pages for more information:

Technical Support - http:/www.nevrona.com/support

Addons, Tips and Tricks and other information - http:/www.nevrona.com/rave

Updates - http:/www.nevrona.com/rave/download.html

Chapter 1
Introduction

In this Section:
• Rave is introduced and related to reporting needs. • Navigation of toolbars is introduced. • Introduction of the main component, the Page. • Brief overviews of the Toolbars, Project Tree Panel, and Property Panel are given.

What's All the RAVE About?

Reporting can be one of the most complex, yet most important tasks for anyone dealing with a database. Reports are the primary visual means to express information retrieved from a body of data. To solve the problems associated with presenting a visual report of data in a meaningful and informative manner, traditional visual reporting tools have offered banded layout tools geared towards table-style listings of data. Today, however, much more complex reporting requirements exist and are not easily handled by banded layout tools.

Welcome to the next level in visual reporting! The Rave visual designer offers many unique features that help to make the reporting process simpler, quicker and more efficient. Rave is an intuitive page based visual design environment that can easily handle a wide variety of report formats, much more than a purely banded style tool. Rave also includes mirroring and other technologies to encourage you to reuse the contents of your reports for quicker changes and easier maintenance. In general though, Rave has been designed to offer the most flexibility and functionality in an easy to learn format.

Where do you begin? Since Rave is page-based designer, many of its features should make sense and be easy to use with only a little practice. There are a lot of options and some might not be obvious when just starting. However, remember that many of these options can be ignored in the beginning, but as your needs and knowledge increases these options are readily available. In fact, Wizards generate "standard" reports without having to know behind-the-scene details. However, we do recommend that you take some time and do a quick read of this manual.

Included with the Rave installs is a project called "RaveDemo" that contains several report samples. To see several common designs, start the RaveDemo.exe and open the RaveDemo project to access the different report types. Exploring the RaveDemo project and other samples are excellent ways to learn Rave.

Report Authoring Visual Environment

A report might be described as data presented in a visual manner, whether it is printed on paper or displayed electronically. Typically, there are sets of database tables that provide data to create a report. For example, suppose there are: a Customer table, a Products table, and an Items Sold table. These tables could be combined to produce form letters, invoices or customer lists.

Let's begin with a quick overview of Rave. The first step to using Rave is to start the program. The first thing seen will be a sheet representing a page on the screen as well as two windows on the side of the page and toolbars across the top of the page. On a first impression, there appears to be many items in the Rave designer, so where do you start?

Let's start with what is shown when Rave is running. First, understand that there are two groups of toolbars displayed in the Rave designer; they are the components and tools.

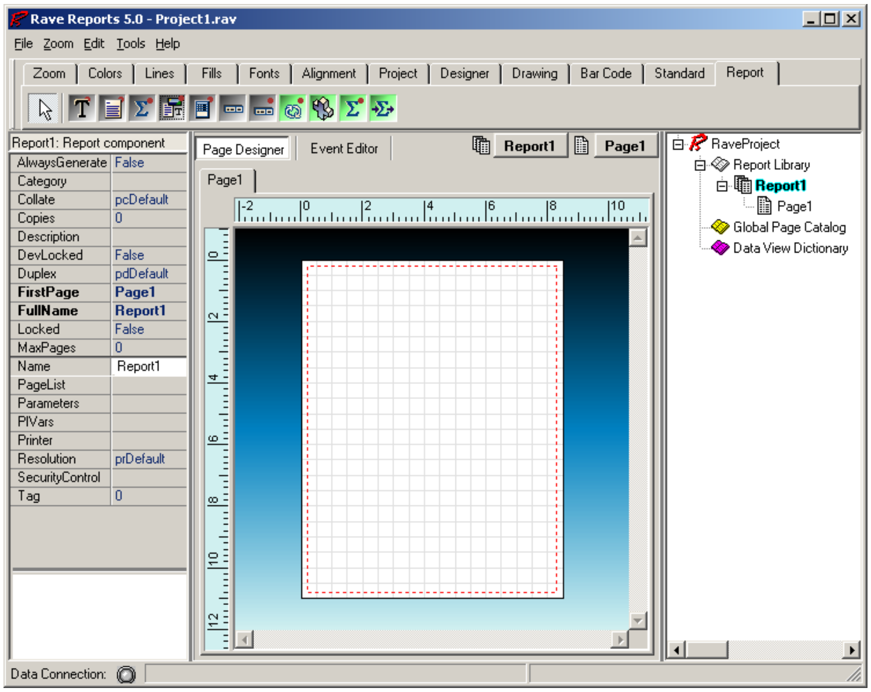
Components are those items that are "dropped" or visible on the page editor. These might be bands, bar codes, lines, shapes, etc. Don't worry about what these things are yet. If the object can be seen on the page layout, it is a component.

Toolbars have the ability to change or modify components, thus distinguishing them as tools. There are several toolbars: alignment, color palette, font editor, etc. If there is a box on the page and it needs to be filled with a background color, first select that item (the box) by clicking on it. Then, use the color tool to change the fill color to any desired color. Toolbars can be hidden and seen by selecting them from the Tools menu. If there is a preference for a more simplified designer look, the toolbars can be hidden until they are needed.

Now that was the very fast overview. There are lots of settings or properties that control the behavior of almost every part of the visual designer. This manual has two main parts, a description of the Rave system including the components and tools, then a reference section that lists the details about each property. Go through the description section first, and then browse the reference section to get a better understanding on just how much control can be had over various design features.

First Glance

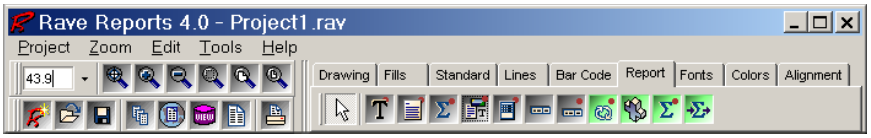
When Rave first starts, it will open a screen that looks like the image below. Since Rave remembers what file was last opened, your screen may look a little different. But, the areas shown are the fundamental pieces. We are going to first explore each part of this screen to give a general tour of the Rave Designer.



Report1: Report component

AlwaysGenerate	False
Category	
Collate	pcDefault
Copies	0
Description	
DevLocked	False
Duplex	pdDefault
FirstPage	**Page1**
FullName	**Report1**
Locked	False
MaxPages	0
Name	Report1
PageList	
Parameters	
PIVars	
Printer	
Resolution	prDefault
SecurityControl	
Tag	0

Navigation Area



The top of the visual designer is the Rave Navigation area. This is where toolbars can be docked individually, and can also be tab docked. Individually docking the toolbars will allow the user to see all the buttons on the toolbar and to see more than one toolbar at a time. Tab docking will allow easy access to each toolbar, but only one toolbar can be seen at a time.

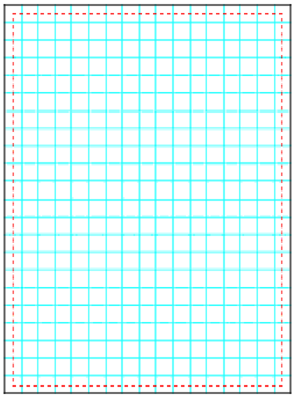
The toolbars contain both tools and components. A tool is a feature that will be used to modify an object already on the page, like the Font Editor or the Color Palette. A component is an object that will be placed on the page like a band, line, text, region, or section.

To find out the name of the button control on a toolbar, simply move the mouse pointer over a button, and a popup window will appear with the name of the tool or component. Also, notice the raising of the button when it is about to be selected. It will become 'depressed' when the button is clicked on.

Each toolbar can also be hidden. To hide or view a toolbar, go to the tool menu and select the desired toolbar from Toolbars menu.

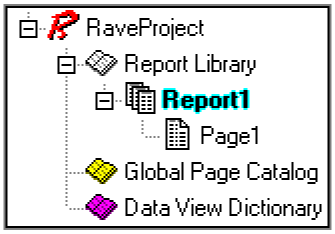
The Page (Foundation of Rave)

The starting point with the Rave Visual Designer is the Page. The page is the foundation and is where all the designing action is done. The Page is represented with the grid pattern, which will look something like the image shown to the right. The look and feel of the Page can be changed with the preference settings, which will be covered in Preferences chapter.



One important thing to know and remember is that the Page has properties, such as height and width. To see or change the Page properties, go to the "Project Tree" Panel and expand the Report node by clicking on the "+" sign, then click on the report name (default is "Report1") and this will show a line that will be defaulted to "Page1". Click once on the Page reference and the Page name will be highlighted in the "Project Tree". The highlight means that the Page has been selected and now the Page properties can be seen in the Property Panel.

Project Tree Panel



The Project Tree panel is a very informative part of the Rave designer and it also provides an easy way to navigate the reporting project structure.

For now just the parts of the Project Tree will be examined. More details are covered in the Project Tree Chapter.

There are three main nodes in the Project Tree: Report Library, Global Page Catalog, and Data View Dictionary. Each of these nodes (and any sub nodes) can be expanded or collapsed by clicking on the plus/minus symbol. Sub-nodes can be created and added, by selecting a desired option (New Report, New Global Page, and New Data Object) from the Project Menu.

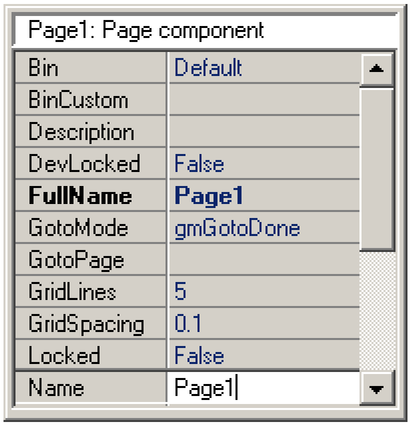
The Report Library node is where all of the reports within the project are contained. Each report will have one or more pages. Each of those pages will normally have one or more components within them.

The Global Page Catalog node is where reporting templates are managed. The reporting templates can contain one or more components. These reporting templates can then be reused via Rave's unique mirroring technology. This could include items such as letter headings and footers, pre-printed forms, watermark designs or complete page definitions that could be the foundation for other reports.

The Data View Dictionary node is where all the data connections for reports are defined. A data view retrieves data from the application through data connections installed within that application.

Property Panel

The Property Panel helps to customize the way components appear or behave. When a component is selected on the page, the Property Panel will reflect the selection by displaying the different properties associated with the selected component.

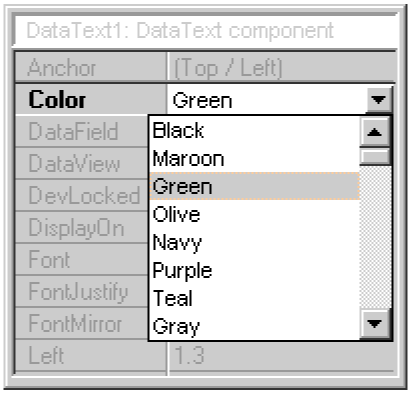


Page1: Page component

Bin	Default
BinCustom	
Description	
DevLocked	False
FullName	**Page1**
GotoMode	gmGotoDone
GotoPage	
GridLines	5
GridSpacing	0.1
Locked	False
Name	Page1

Changing the properties values is easily done by using various associated drop-down menus and edit boxes. If no component is selected, then the property panel will appear blank with no options to choose from.

Another way to change a property is select a value from a list of possible choices. For example, the Color property has a down arrow button. Clicking on the down arrow button will display a list of colors that can be selected. Any property that has a list of choices can also be double-clicked (instead of clicking on the down arrow button and selecting the option) to advance to the next item in the list.



DataText1: DataText component

Anchor	(Top / Left)
Color	Green
DataField	
DataView	
DevLocked	
DisplayOn	
Font	
FontJustify	
FontMirror	
Left	1.3

Chapter 2
Toolbars and Tool Windows

In this Section:

- General information about the toolbars
- Toolbar placement in the Visual Designer environment
- How to manipulate Toolbars within the environment
- Introduction to Tool Windows and their movement

Toolbars

A toolbar is a collection of icons that relate in their function. Rave has several toolbars that can be docked to any side of the designer window, tab docked, or hidden to stay out of the way until they are needed.

To access the toolbars, go to the Tools menu, and then from Toolbars select the toolbar that is desired. Selecting a toolbar will place a check mark next to the toolbar.

The toolbars are designed to remember their last placement position. When a toolbar is closed and then opened again, the toolbar will be in the same position on the screen that it was in before it was closed.

Toolbar Placement

When a toolbar is visible it can be in one of two states: tab-docked or single docked. A tab-docked toolbar is docked with tabs displayed. A single docked toolbar has a ribbed section on the left of the toolbar. To change the toolbar state, right-click on one of its icons. This will reveal a small button with either a "dock" or "undock" label. Click on that button to change that toolbar's state.

When a toolbar is undocked, the "toolbar handle" is the ribbed section in docked style. To move a toolbar, place the mouse cursor somewhere on the "toolbar handle", press and hold the left mouse button while dragging the toolbar to a new location in the top portion of the designer window. The best way to get familiar with this is to try docking and undocking toolbars.

Below is an example of docked and tab docked. As previously mentioned, the single docked toolbars can only be placed at the top of the designer window. The user can practice docking and undocking the toolbars to get familiar with this process.

Toolbar Palette

The toolbar palette has a number of special features, which are worth mentioning in more detail. The region on the upper right hand corner of the visual designer and allows one or more toolbars to be docked. When the toolbars are docked, tabs are formed with the name of the toolbar as the name of the tab. As toolbars are placed in the tab docking area, a new tab will be created to reflect the current toolbar.

There is a tab grabber that allows the tabs to be moved to the left or to the right. When the tabs are moved to the far right, double arrows appear to allow movement through the tabs. The entire tabs area can be emptied by moving all the toolbars away from the tab area, or by closing the toolbars from the tool menu. The tab area can also be moved off to the every edge of the screen (moved to the far right) to allow room in the upper area of the visual designer. Use the Tab Grabber to move the Tab area (it can be filled or emptied) off to the far right.

Hiding Toolbars

There are a couple ways to hide toolbars. Hiding Toolbars allow the user to 'clean' up the Rave Designer. This is also helpful to get unused toolbars, or less frequently used toolbars, out of the way or out of the designer environment.

The method to do this is to use the Tools menu to uncheck, and thus hide, the toolbar.

Component vs. Utility Toolbars

There are two types of toolbars: Component toolbars and Utility toolbars.

The component toolbars are used for placing components that work and have function within a report; see the Components Overview Chapter for more details about components. Most of the components appear visually when placed on the page, however there are a few components that are non-visual. Non-visual means that the components only appear in the Project Tree, there will be no visual record of the component on the Page.

Utility toolbars are used to set properties that affect the look of components in a report. For example, the Utility Toolbar Color can change the color of text in a DataText component.

Below is a list of the toolbars classified by type:

Component Toolbars	**Utility Toolbars**
Standard	Alignment
Report	Colors
Bar Code	Fills
Drawing	Fonts
	Lines
	Zoom
	Project
	Designer

Tool Windows

Rave has two very helpful windows, which are generally called the Tool Windows. They are called this because the user will use them as tools to complete and navigate a report. The two Tool Windows are the Project Tree and the Property Panel.

These windows will most likely be displayed when Rave is started. But, if it is not displayed, then go to the Tools menu and select these windows to be opened by placing a check mark in front of them.

Property Panel

The Property Panel gives information about the characteristics of a project and the components within the project. It will be most helpful when designing reports because it gives easy access to component properties, and thus easy visual access to modifying the properties. More detailed information is given in the Property Panel Chapter.

The Property Panel displays and allows modification of values of any component that is placed on the Page, which includes non-visual components. To get property information about a certain component, remember to select that component before looking at the Property Panel (see Selecting a Component in the Page Designer Chapter for more information on how to select an object).

Project Tree

The Project Tree is a very informative part of the Rave designer. It provides an easy way to navigate the reporting project structure.

The Project Tree is composed of three main branches: Report Library, Global Page Catalog and Data View Dictionary. For more information about the functionality of each of these components, please see the Project Tree Panel chapter and the Project Components chapter.

Items within the Project Tree can be selected by clicking on the item. Also, remember that the components seen in the tree are also the components that are on the Page.

Note: the non-visual components are in the Project Tree and are usually a green color.

The details of navigating through the Project Tree are saved for a chapter in itself. For now just a general overview of the Project Tree is given.

Chapter 3
Components Overview

In this Section:

- Introduction to Components in general
- Lists Component Toolbars
- Overview of each Component toolbar is given

What is a Component

A component is defined as something placed on the page, such as a barcode, line, region, shape, etc. The components available in Rave can be found on any of the component toolbars (e.g. Standard, Drawing, Report and Barcode).

The toolbars are made available by clicking on the Tools menu followed by the Toolbars menu. The available toolbars will then be shown in another submenu and will have checkmarks showing the toolbars that are currently visible. Once a component toolbar is active and selected, a component can be selected and placed on the page. The Page is a special base component, and more details are given in the Page Designer chapter.

Special properties are associated with each component. These component properties can be seen using the Property Panel. Set the properties of each component to the desired setting by either typing the setting in a text dialog, using a drop down menu, or by using the special ellipse (…) button to get to the property dialog box.

There are many properties associated with each component, but don't be intimidated by the number of properties. The properties are there to allow adjustment for a component's behavior and in many cases the default settings are adequate. Also, please note that the number of properties listed with each component may vary depending on the user level that has been set under preferences. To adjust the user's level, please visit the environment tab in the preferences dialog. See the Preferences chapter for more details.

Since there are many properties associated with each component, this chapter will focus mainly on the component toolbars rather than their associated properties. The current chapter provides a good overview of what each component toolbar does without too much detail about the property specifics. Do note that many components share common properties, so once the common properties are learned for one component, they can be applied to other properties.

Components are also defined by their relationship relative to other components. This relationship is defined by a parent-child relationship. When a text component is placed on the page, the parent is the page component and the child is the text component. Another way to look at it is that the page contains the text component, thus the parent component contains the child component.

The parent-child relationship also extends into the positioning of the components. All positions are relative to the upper left corner of the parent, thus the Left property and Top property are used to define the relative position of a component. If the parent is like a Section component, which can contain any number of other components; then as the parent component is moved around, it's children components will move accordingly. If the parent component is deleted, all of its children will also be deleted.

A parent-child relationship is defined when the child component is initially created. If the child component is dropped within the parent component, then it will become the child of that parent. If the component is dropped on the page, then dragged on top of another component (such as a

Band or Section), it will still be considered a child of the page.

The Project Tree, which will be explained in more detail in a later chapter, visually shows the parent-child relationships of the components. In general, the Project component is the master parent of all reports, global pages and data objects. Reports are parents to the report pages. Pages are parents to the components that are placed on them. There are still other components that can be parents to components, such as Regions, Bands and Sections.

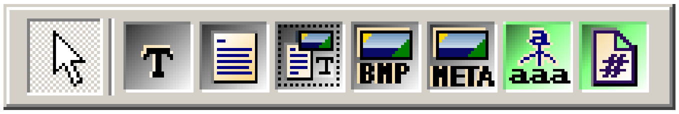
The Component Toolbars

There are four standard component toolbars. Other component toolbars may be present if add-on packages have been included. The normal set of component toolbars is: "Standard", "Drawing", "Report", and "Bar Code". There is one particular property that applies to all components, so it will be mentioned here.

The "Name" property is used to assign a new name to that control or to designate the name of the control to give it a unique meaning. By default, Rave assigns sequential names based on the type of the component, such as 'Section1', 'Section2', and so on. Change these to more meaningful names to make the Project Tree more readable and the reporting project easier to maintain. The "Name" property must not contain any spaces or special characters. This is the name that will be used in the Project Tree Panel.

The next several sections will give an overview of the four component toolbars. This will serve as an introduction, and more detail will be covered in later chapters.

Standard Components



The Standard Components toolbar controls the visibility of most frequently used components. Once the Standard components toolbar is active (either docked or undocked in the Rave environment), select and place components such as a bitmap, section, text control, etc. on the page.

The Text component creates a single line of text, which can be changed using the Property Panel.

The Memo component allows several lines of text to be displayed, including multi-line text.

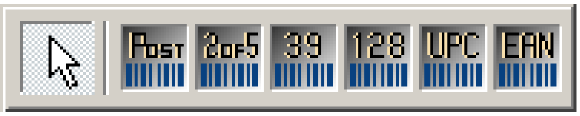
The Section component is a very special component that allows several components to be grouped together.

Bitmap and Metafile components allow images to be placed onto the report. The FontMaster component allows the user to define standard fonts for different parts of the report, like the headers, body, and footers.

The final component is called PageNumInit, this component allows the restart of page numbering within the report.

Use these components to create the base of a report, by inserting text and images.

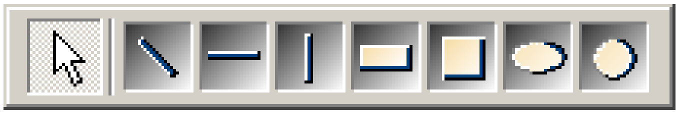
Bar Code Components



The BarCode Components toolbar contains six bar code components: PostNetBarCode, 2of5BarCode, Code39BarCode, 128BarCode, UPCBarCode, and EANBarCode.

These are used for making all sorts of bar codes for report usage.

Drawing Components

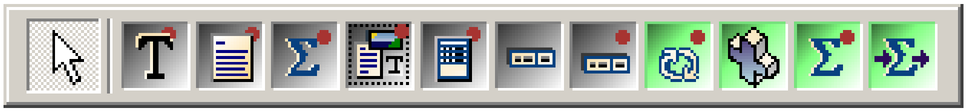


The Drawing toolbar controls all graphical drawing components. This includes drawing lines, boxes, rectangles, circles and elliptical circles.

The components from left to right are: Line, HLine, VLine, Rectangle, Square, Ellipse, and Circle.

These are used for creating graphical images and general report formatting. Use these to separate areas of a report, or to create an informative image.

Report Components



There are several report components: (left to right) DataText, DataMemo, CalcText, DataMirrorSection, Region, Band, DataBand, DataCycle, CalcOp, CalcTotal, CalcController. Use these components to interact with a database and make a functional report.

The components with a red dot in right corner are Data Aware and are capable of displaying information from a database. Each component has a DataView property, which allows the component to interact with a database.

The components with a green color are "non-visual". These can be seen in the Project Tree, but not on the Page layout.

Bands and DataBands must be placed within a Region. Therefore before placing a band on the Page, first place a Region in the area in which a Band or a DataBand will be established. There is no limit on the number of Bands within a region, nor is there a limit to a single region. There can be as many regions as needed. Each Band has its own set of properties to control its behavior. See the Report Components Chapter for more detailed information.

Chapter 4

Page Designer

In this Section:

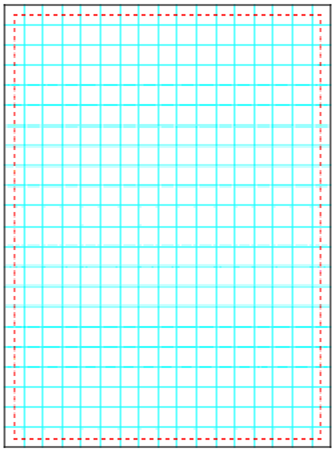
- Introduce the Page as a basic component
- Discuss how the Project Tree and Property Panel relates to the Page
- Show how to select component(s)
- Learn how to re-size and move components
- Overview of cutting and pasting in Rave

Overview

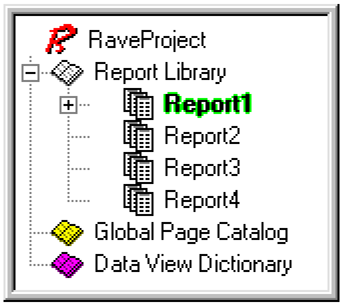
Just knowing the components and tools is not enough to work effectively in Rave. One must also learn about the most fundamental component of Rave, which is the Page. In this chapter the Page and its relation to the Project Tree and Property Panel are explained in detail. Although the Project Tree and the Property Panel have their own chapter, as each are important individually, this chapter explains how each is related to the Page. Also in this chapter, some basics on moving objects on the Page are given.

The Basic Component: The Page & it's Panels

The base component of the Rave Visual Designer is the Page. The Page is not a normal component in that it is not on a Component Toolbar, but it does have properties and it will help to think of it as the base component that provides the container for the report design. All Rave components must be placed on the Page. The Page is represented by the grid pattern on the page panel. The look and feel of the Page is controlled by preference settings that are covered in the Preferences Chapter. Changing the visual look of the Page is discussed in the Utility Toolbars Chapter in the Designer Toolbar section.



Like all components, the Page also has properties such as height, width, description and name. To see or change the Page properties, go to the "Project Tree" panel and expand the Report node by clicking on the "+" sign, then click on the report name ("Report1" is the default report). Clicking on the report name will show "Page1", which is the default name for a Page. Click once on this and notice that *"Page1"* will be highlighted. This means that the Page has been selected and now the Page properties for that Page will be seen in the Property Panel.



Page properties allow a report to be more informative. For example, rather than have a report with Page1, Page2, Page3, it might be desirable to have the pages named Invoice, PO, PackingSlip. To do this, use the "Name" property in the Property Panel. The name can be anything the user desires, but note that the name cannot have any spaces or special characters. Remember the Name is for the Rave designer only and is visible only in the Project Tree and on the tab(s) of the Page panel.

In the image of the Project Tree and Property Panel, the page names have been changed to show how useful it is to rename the Page. Now the project tree looks more informative. It is useful and helpful to adjust any properties associated with the Page to make the report more informative and useful for future use and revisions.

Selecting Components

Selecting components is as easy as clicking on them. Moving the mouse pointer to the region where the component is and clicking the left mouse button once will select any component that is placed on the Page. A selected component will appear surrounded with a line that has colored dots, called pips.

The "pips" can be of four different colors: green, gray, red, and yellow. Green pips indicate that the current component is the only one selected. Gray pips show that there is more than one component selected, and red pips indicate that the current component is locked. The yellow pips indicate objects that are that have been mirrored and cannot be modified.

Selecting more than one component at a time is done in a similar way to selecting one component. The difference is that when selecting multiple components; hold down the Shift key while clicking on the objects.

Another way to select either one or more components is to draw a "virtual" box around them. To do this, select an area that is adjacent to the component and drag the mouse to surround the component(s) keeping the left mouse button clicked.

To unselect a component, simply click on the Page Designer tab or move the cursor selection to a different component by clicking on it.

When multiple components are selected, only the common properties will be displayed in the Property Panel. When a property changes, the change will affect all the selected components at that time.

There is a special property for all components called Locked. If the Locked property is True, the component can be selected, but it's properties cannot be changed (except for the Locked property, of course). If selected components are locked, then the pips and the property name for those items will be red (as mentioned previously). If the component is a child component to a parent component, the parent will control the locked access. The Locked property state drop-down menu will show either a True/False indicating the lock status or it will show the parent component name that is controlling its lock state. The remaining discussion is about components that are NOT locked.

Sizing and Moving Components

To move a component, simply click on it with the left mouse button and drag it to the desired area. When more than one component has been selected, they can all be moved by just moving one of the components. Note that this will NOT affect components that have the Locked property set to True.

The main purpose of the "pips" that appear when selecting a component are for resizing. Resizing only works on ONE component at a time, i.e., a group of selected components cannot be resized at one time. A component can either be resized vertically, horizontally or diagonally. If the resizing is done diagonally, the component will resize both vertically and horizontally. Simply click on one of the pips with the left mouse button and drag the pip until the desired position. Locked components cannot be resized.

Cutting and Pasting

Cutting and pasting is achieved in the same way as in any other standard Windows applications. After selecting a component (or group of components), choose Cut or Copy from the Edit menu; or use the Ctrl-X or Ctrl-C key combinations; or right click on the component and selecting Cut or Copy from the popup menu.

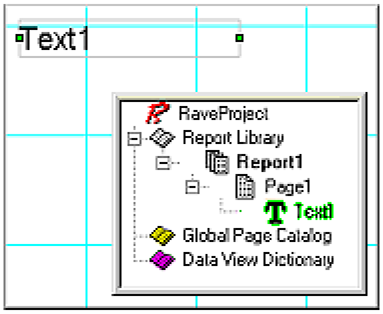
To paste the components, choose from any of the three previous methods (Edit menu, Ctrl-V or popup menu).

Again it is important to point out that Locked components cannot be cut since this would involve removing the object from its position.

Exercise: Selecting, Sizing, and Moving Components

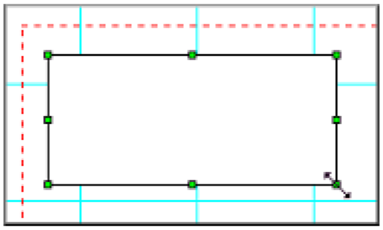
To select a component:

1. Click on the component. The pips and border will appear around it
2. Or, click on the component in the Project Tree, and notice that the selected component will be highlighted.



To resize a component:

1. Select the component
2. On a pip, place cursor over it until it changes to a double arrow
3. Click and hold the pip, then drag to the desired size



To move a component:

1. Select component
2. Move cursor over component (not near a pip, as that would be resizing the component). Click on component and hold
3. Move component by dragging cursor, and thus the component
4. Place in desired location.
5. Release cursor

Exercise: Cutting and Pasting

Using Edit menu:

1. Select the component.
2. Select Cut from the Edit menu. This will place the component into the clipboard, and the component will disappear visually from the page.
3. Paste by selecting Paste from the Edit menu.

Using the short-cut keys:

1. Select the component.
2. Press Ctrl-key and X-key. This will place the component into the clipboard, and the component will disappear visually from the page.
3. Paste by pressing Ctrl-key and V-key.

Using the pop-up menu:

1. Select the component.
2. Right-click on the component, in the pop-menu select Cut. This will place the component into the clipboard, and the component will disappear visually from the page.
3. Paste using one of the two previously mentioned ways.

Chapter 5

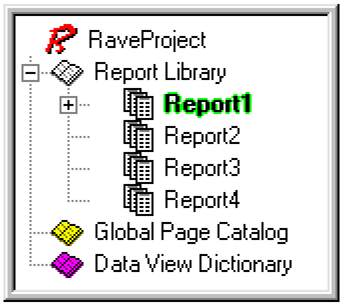
Project Tree Panel

In this Section:

- Project Tree structure introduced
- General discussion of Parent-Child relationships
- Introduce the main project categories
- Show How to Load/Unload Global Pages
- Discuss and explain the Drag and Drop features

Overview

The Project Tree Panel provides an easy way to navigate through the report design structure. This tree style view gives an outline overview of the report pages and the report structure. Thus, this makes the Project Tree a very informative part of the Rave designer. Once you have designed some simple or complex reports, remember to visit this panel and see just how informative it can be.



A project can contain multiple definitions within each main category: Report Library, Global Page Catalog and Data View Dictionary. Each node (and any sub nodes) can be expanded or collapsed by clicking on the plus/minus symbol. Any of these main nodes can be added from the "Project" menu option. There you will see an option for each of the nodes "New Report", "New Global Page" and "New Data View". In order to add a "New Page" to a Report in the Report Library, activate the Report that and use "New Page" from the Project menu.

Expanding and Right-Clicking

An easy way to complete tasks on items in the project tree is to right-click on the item. There are many different pop-up menus available. Clicking on a Main Node like the Report Library brings up a popup menu with two choices: Expand All and Collapse All. These two choices refer to expanding and collapsing all the nodes, which would provide easy fast access to all items in that tree.

A right-click on a sub-node, like on a Page, would reveal the option of deleting a page. Some of the available functions may be few or many, as in the next image.

Right-clicking on a component can display a pop up menu with many options. These options are the same options available as if a right-click was completed on the component on the designer page. Also, these items reflect the options available to the component through the Utility toolbars.

There are many options that may be available with a right-click of the mouse. To speed up the report making process or to make navigating a report fast and easy try to become familiar with the options that are available through these popup menus.

Parent-Child Relationship

Now, that you have seen how to click through the Project Tree, we'll review some lingo that is associated with a "tree".

The Project Tree is listed in a way that looks like an upside down tree. The base or the root of the tree is at the top of the list. Each subsequent branch is a child of the root. This makes the root also the parent of the child or branch. Each branch can also be its own parent and have its own child or children.

The techniques used in the previous section demonstrated how to move through the tree by expanding and collapsing the nodes associated with each parent. When a branch has children, a node will appear to the left of the branch name.

The Report Library is the root of the Tree, and has one child called Report1, shown with the text highlighted.

Report1 has several children: Page1, Page2, Page3, Page4, Page5, and Page6.

Page1 has the following children: Text1, Text2, and Memo1.

Also there are two sibling groups. Just like in a real family with parents and children, the children can have siblings. The first group of siblings: Page1, Page2, Page3, Page4, Page5, and Page6. The second group of siblings: Text1, Text2, and Memo1. There are other family associations, but the main two that we are concerned with is the Parent-Child relationship.

Report Library

The Report Library node is where the design structure, including all the components used, in each of the report(s) will be displayed. Normally, a component is selected via the Page in the Visual Designer layout panel by simply clicking on the component. However, the Report Library tree can also be used to select components by simply clicking on the component name. When a component name is clicked, that name will be highlighted. To select multiple components, hold the Shift- key down while selecting as many components as you like in the Project Tree.

A project can be simple with one or two report definitions, or it can be complex with many report definitions. Each report that is created would have it's own design structure. To select and switch to a different report, simply double-click on that report name in the Report Library tree structure.

TIP: To make navigation and understanding of a report easier, make use of the Name property. See the Name Exercise at the end of this chapter for help in using this property effectively.

Global Page Catalog

The Global Page Catalog node is where Page definitions, which contain templates to be mirrored, are kept. This includes things like Letterheads, Forms, Watermark designs, and other Page definitions that could be the foundation for several reports. This can even be a complete report design. An example would be mirroring a section component on a Global Page where the same contents (like an "Invoice") could be printed, but also with a different caption at the bottom of the pages like "Original", "File Copy", and "Shipping".

Loading and Unloading Global Pages

Global Pages are very useful and helpful to the designer, especially when they can be accessed quickly. These pages can hold all sorts of objects and components that can be repeated and mirrored throughout many pages in many different reports. Many of the reports in a Report Library could make use of any of the Global Pages in the Global Page Catalog.

When creating reports, a useful feature of the Rave Designer is the Page Designer Tabs. These are located at the top of the Page Designer window. They are the tabs of pages in use.

Global Pages can be added to this tab for quick referencing, meaning that the designer does not need to scroll through the whole Project Tree to find the right Global Pages frequently needed. Instead the designer just needs to load the page to the Page Designer Tab area and click on the tab when it is needed. Global Pages can be loaded and unloaded to/from the Page Designer Tab area by performing a right-click on the Global Page and using the speed menu to make their choice. A fast way to unload a Global Page is to select it from the tab area, then click on the Page and use the Ctrl-F4 keystroke.

Data View Dictionary

The Data View Dictionary node in the Rave Designer is where all the data connections for the reports would be listed. To add a Data View to this list, select a "New Data View" (from the "Project" menu). This will bring up a "Data Connections" dialog window. This window will show all the Rave data connections that are "active". Then select any of these and this will create a new Data View attached to that data connection.

Ctrl- (Control) Drag and Drop

Ctrl-Drag and Drop is used to create or mirror an item from the Project Tree structure either to another age node in that report structure or to the current page on the designer layout. When dragging from the Project Tree Panel to the page designer, the drop location will be the location where the mouse button is released. However, when dragging from one page node to another (in the Project Tree Panel), the location of the drop will default to the same place as the source.

NOTE:
When Ctrl-dragging a non-visual component, it must be one that can be mirrored. Copying a component that cannot be mirrored will NOT work. Regions and Bands are components that do not enable mirroring. To mirror a region, make the region part of a section, and then mirror that section. The section could include other components as well.

CAUTION:
Drag and drop will work from Report Page to Report Page, Global Page to Global Page, Global to Report Page, however, it will NOT work from the Report Page to Global Page, since components on Reports are not visible to other Reports and therefore cannot be used as mirrors on a Global Page.

Alt-Drag

The Alt-Drag is used to change the parent of a component to a new "container" component like a Section. All moves of that component need to be within that same Page design. When a new Section is created, the user may choose to have items from another Section placed in the new Section. In this case the parent must be changed from the old Section to the new Section and then the new Section can be mirrored.

NOTE:
The target component must be a container type. The Page, Region and Section are all holders or containers. So, if there are two or more sections on a Page, items can be dragged from one Section to the other Section. Also components can be dragged from the Page to one of the Sections.

Exercise: Navigating the Project Tree

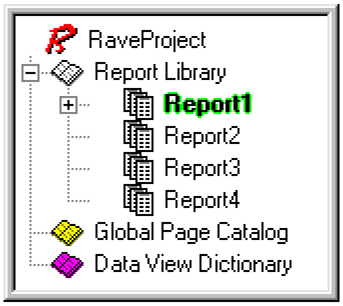
Because many report projects are complex, the Project Tree's hierarchy helps to navigate through the reports.

To navigate through the hierarchy (expand, collapse, and selection):

1. Expand an item by clicking the square icon with a "Plus" symbol. This will change the

item to a "Minus" symbol and expand the items into the sub-items such as Report1, Page1, etc., depending on which level of expansion is being expanded.

2. Collapse an item by clicking on the "Minus" Symbol. This will change the icon to a "Plus" symbol and collapse the items, so that they will no longer be visible.
3. Select an item in the Project Tree by clicking on the item. Selection will be indicated by that item being highlighted.



Exercise: Naming Components

Since projects can become very complex, it is highly suggested that the "Name" property be used to make items in the Project Tree easier to understand.

By default, the "Name" property will automatically default to something like "Report1", "Report2", and so on. However, it would be easier to maintain a project if the reports had useful names. This can be done through the "Name" property.

When developing a naming convention, be creative, but remember **no spaces or special characters** are allowed in the "Name" property.

In the example Project Tree Panel, the default report names have been changed. Instead of dealing with reports listed as "Report1" through "Report7", the reports were labeled accordingly: "CustomerList", "CustomerLabels", "CustomerDue", "Invoice", "PO", "ProductsOnHand", and "ProductsOnOrder". It is important to understand that the Project Tree Panel is there to make overseeing a project easier. But, without an informative naming system, the Project Tree Panel will not live up to it's full potential.

To rename a report (or any other component):

1. Select the report (component)
2. Ensure that the report to be renamed has been selected. A selected item is highlighted.
3. Go to the Property Panel and look at the Name Property.
4. In the "Name" Property edit box, type in the desired name.
5. Hit Enter or click somewhere out of the edit box, after a new name has been entered.
6. Look at the Project Tree and notice that the name of the report (component) has been changed from the default name.

Exercise: Loading and Unloading Global Pages

Loading and Unloading Global Pages to the Page Designer Tab area makes referencing these pages fast and helpful. This is especially true when dealing with mirroring sections and objects from Global Pages to the pages in different Reports.

To load a Global Page:

1. Start with a new report
2. Insert about 4 blank reports, by selecting **New Report** from the **Project** menu or by clicking on **New Report** from the Project toolbar
3. Next, select "Page1" in the first report. Do this by using the Project Tree. Click on "Page1" in the tree or click on the "Page 1" tab in the Page Designer Tab area. Once that Page has been selected, it's name will be highlighted.
4. Add about three Global Pages to the report. Do this by selecting **New Global Page** from the **Project** menu or by clicking on **New Global Page** from the Project toolbar
5. Notice that since the first page in the first report was selected when the Global Pages were added, the new Global Pages were added to the Page Designer Tab area. The pages are also added to the Project Tree under the Global Page Catalog
6. Click on a different Report page. Notice that once a different page is selected, the Page Designer Tab area only has that page in the Page Designer Tab area
7. To add Global Pages to the Page Designer Tab area right-click on a Global Page in the Project Tree and in the menu that pops **up select** Load Page. Try loading all three Global Pages
8. As the Global Pages are loaded they will be seen in the Page Designer Tab area
9. Try loading Global Pages to other pages in the last two reports

To unload a Global Page:

1. Complete the steps from the previous exercise
2. Select a page from Report 3 that contains all the loaded Global Pages
3. There are two ways to unload a page. One way is to use the speed menu; the other is to use quick keys
4. The first way, using the speed menu, requires that you first select the page that needs Global Pages unloaded
5. Next, right-click on the Global Page and **select Unload** Page from **the** speed menu. Do this for **all three** Global Pages in Report 3
6. We will use the second way to unload Global Pages next
7. Select the next page in "Report 4". Make sure it contains all the Global Pages in the Page Designer Tab area
8. Once the page in "Report 4" is selected and you can see the three Global Pages in the Page Designer Tab area, select one of the Global Pages from the Page Designer Tab area
9. Now use the **Ctrl-F4** keystroke to unload the Global Page
10. Try this with the rest of the Global Pages in the Global Page Catalog.

Exercise: Dragging a Component (Ctrl-Drag)

One of the more common items to drag would be to copy a Data Field over to the Page in the Visual Designer. The target could be an existing DataBand or just an open area on the Page. The Data Field will be converted to a DataText or DataMemo on the target area depending on the source's field type. Accomplish this by doing the following steps.

To drag a component:

1. Go to the "Data View Dictionary" node, expand it if it is collapsed
2. Select one of the Data Views and expand it if it is collapsed
3. Select (highlight) a field component to copy
4. Press and hold the Ctrl-key, now drag that field over to the Page

5. Release the mouse and Ctrl-key when it is at the desired location
6. Select the newly copied data field and set its properties as required

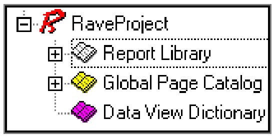
This exercise may have been a little advanced for this point in the manual, but it is put here to give the user the steps necessary to complete a dragging component. So, since a full report has not yet been created, don't worry about getting the report working yet.

Exercise: Changing the Parent of a Component (Alt-Drag)

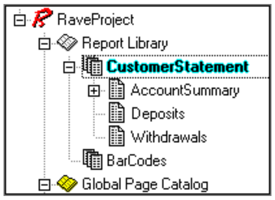
Changing the parent of a component can be very helpful, like when trying to group or ungroup components. For example, say there was a Page that contained components that would be better grouped together into a Section. The current status is that the Page is the parent component of the components that need to be grouped. To group the items together, the parent component would have to be changed to the section component. After this is done, the items will be in a section, which can then be moved around as such. In the reverse sense, maybe a component needs to be taken out of a section and needs to just lay on the Page and not the section, just use the Alt-Drag to complete this task.

Changing the parent of a component:

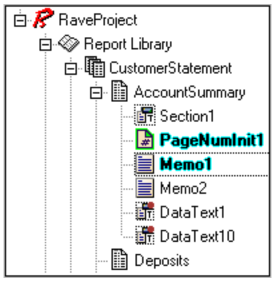
1. Go to the "Report Library" node; expand it if it is collapsed



2. Expand the Report node that contains the Page definition needed

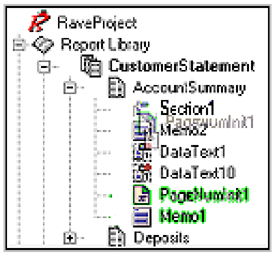


3. Expand the Report node so that both the source and target component can be seen. In Customer Statement, there is a Section component already on the Page. We will change the parent of PageNumInit and Memo1 from the Page to the Section



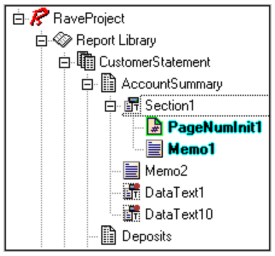
4. Select the components that will have their parent's changed. To select multiple components, hold the **Shift** key while clicking on each component. When done selecting release the **Shift** key. The desired components should be highlighted
5. Press and hold the Alt key, now drag the source components to the new parent component. In this example, the Alt key is held while selecting and dragging either of the two components. While dragging the components to the new parent, notice that cursor changes as it is placed over the target. When the cursor

indicates it is over the target, release the mouse button, then release the **Alt** key



RaveProject
Report Library
CustomerStatement
AccountSummary
Section1
Memo2
PageNumInit1
DataText1
DataText10
PageNumInit1
Memo1
Deposits

6. Notice that the components are now under the new parent and that the parent has a "minus" sign next to it to indicate that it is the parent of some components



RaveProject
Report Library
CustomerStatement
AccountSummary
Section1
PageNumInit1
Memo1
Memo2
DataText1
DataText10
Deposits

7. When done, the newly moved components can be set as .
required

Chapter 6

Property Panel

In this Section:

- Defines and describes the Property Panel
- Explains the different types of properties
- Describes the property editor
- Covers the purpose and items in the speed menu

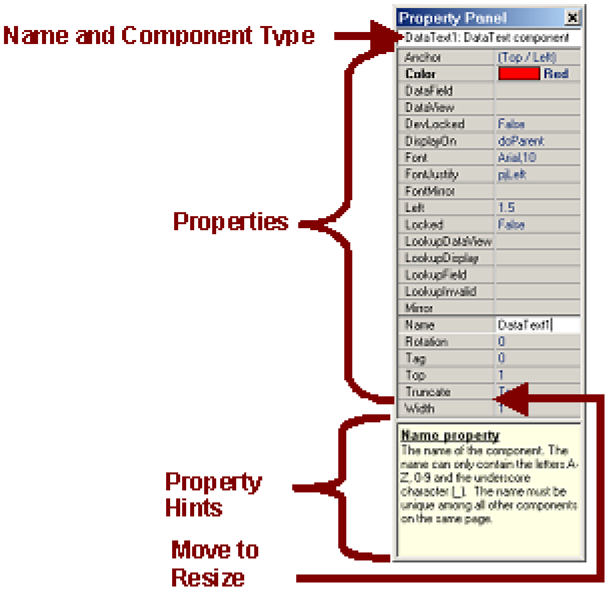
What is the Property Panel

The Property Panel gives information about the project components. It will be most helpful when designing reports because it gives an easy access to all properties associated with each component in a report.

The Property Panel displays and allows modification of values of any component that is placed on the Page. To get property information about a certain component, remember to select that component before looking at the Property Panel (see Selecting a Component in the Page Designer Chapter for more information on how to select an object).

The Property Panel displays the current properties of a component. For example, by clicking on a DataText component, the component will be selected and the Property Panel will display all the properties that are available for the component.

There are three areas to the Property Panel. The first is the very top, which gives the name and type of the component. The second area is the middle, where all the component properties are listed into two columns. The left column displays the property names and the right column displays the current value of those properties. At the bottom of the Property Panel is the third area. A Property Hints box, which gives a brief explanation of the currently selected property, is located in this third area.



Property Panel

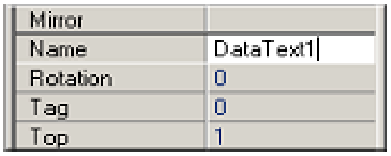
DataText1: DataText component

Anchor	(Top / Left)
Color	🟥 Red
DataField	
DataView	
DevLocked	False
DisplayOn	doParent
Font	Arial,10
FontJustify	pjLeft
FontMirror	
Left	1.5
Locked	False
LookupDataView	
LookupDisplay	
LookupField	
LookupInvalid	
Mirror	
Name	DataText1
Rotation	0
Tag	0
Top	1
Truncate	T
Width	1

Name property
The name of the component. The name can only contain the letters A-Z, 0-9 and the underscore character (_). The name must be unique among all other components on the same page.

In the property listing section (middle), the line that separates the two columns can be moved to resize the columns to allow for better viewing of the names/values. To do this, move the cursor over the line until it changes into the double arrows. Press the left mouse and hold, then drag either left or right to resize the columns.

Remember a component must be selected before the Property Panel will reflect that component's properties. It is also possible to select multiple components at the same time. Once a component is selected, any of the properties in the Panel can be selected by clicking on the desired property name or it's adjacent box selection. With the property selected, it is then possible to change the property value. When any of the fields in the Property Panel are selected, the up/down arrow keys can be used to move up and down between properties.



Mirror	
Name	DataText1
Rotation	0
Tag	0
Top	1

Some of the property names appear in bold. This indicates that the current value differs from the default value. For example, when a DataText component is dropped onto the Page, the default color is Black. By changing the color to something other than Black, like Red, the property name "Color" changes to bold. When the setting is set back to the default value, the bolding will be removed.

Of what use is the bolding? Let's say for a moment that a report that has been designed by a colleague has a problem and it doesn't work. By quickly glancing at the changed properties, this will show what properties have been changed, which might in turn reveal the report functionality problem.

While most components can be selected by clicking directly on them, the Page is a special case. Clicking on the Page itself in the Visual Designer window will clear the Property Panel of all properties, thus not selecting the Page.

To select the properties of the Page, click on the tab at the top (in the area called the Page Designer Tab area) where it shows the page number. Another alternative is to select the Page by clicking on it from the Project Tree.

Types of Properties

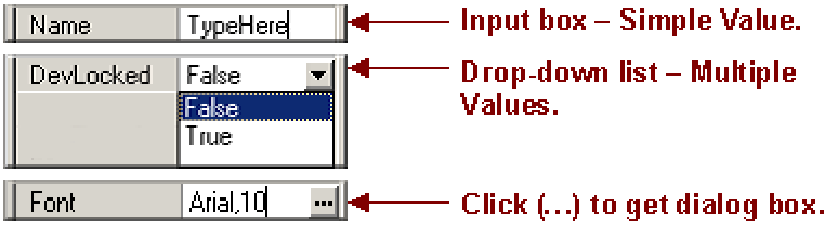
Every property has an associated type. These types can be numerous, different, and can be edited in the Property Panel. The property types can be string, or integer values.

When editing a string property, it is valid to enter any character that can be displayed on the screen. When editing integer values, it is valid to type any numeric value. Decimal numbers are another common property that may be edited.

Often times there are properties that have predefined values that may be selected. These values are displayed in the form of lists containing the valid items that can be selected. There are also properties that have themselves a number of different properties associated with them.

Property Editors

Depending on the type of property selected, there are three types of input boxes available. The first one is a simple input box where a value is typed. The value in the input box can be either numeric, alpha, alphanumeric or one of a set of possible values. A drop down list is the second type of input box a user can encounter. If the values of a property are from a set of previously defined values, a dropdown list will appear by clicking the small down arrow that appears on the right-hand side of the box. Select the desired value by clicking on it. The third option is a dialog box. Some properties require more than one value to set the property (such as fonts). With these properties, a button with three small dots (called an ellipse) appears on the right-hand corner of the property edit box. By clicking on the dots, a dialog box (or wizard) will appear and from this dialog box the multiple properties can be set.



Name TypeHere

DevLocked False
False
True

Font Arial,10

Input box – Simple Value.

Drop-down list – Multiple Values.

Click (...) to get dialog box.

Right-Click Menu

A popup menu will appear when the Property Panel is right-clicked on. This speed menu can be used to set various options of the Property Panel.

Property Hints, the first menu item, hides or shows the Property Hints box which is located at the very bottom of the Property Panel. Highlight Changes, the second menu item, enables or disables the showing of the properties in bold for items that have been changed. Exclude Name, Size and Pos Changes, the third menu item, determines whether Name, Size and position changes (like width and height) are included in the properties that will be bolded when the default value has been changed. The third menu item becomes helpful when many properties have been changed, and there is a need to see changed properties that are not commonly changed.

Exercise: Navigating the Property Panel

Opening the Property Panel

1. If the Property Panel is not visible, then go to the Tools Menu. From the **Tools** sub-menu, choose **Property Panel.** Make sure that there is a check mark next to the selection.

Identifying Property Panel Areas

2. For this example, the base component, the Page, will be used. So, to select the Page component, click on the page tab in the Page Designer Tab area. Notice that after selecting the Page, its properties are displayed in the Property Panel.
3. Next make sure that the Property Hints box is visible at the bottom of the Property Panel, if it isn't already. Right-click anywhere in the Property Panel. A pop-up menu will appear. In this menu, check all the options available.
4. Expand the Property Panel out so that you can see all the properties as well as the Property Hints box. On steps 5 to 8, read the Property Hints box to find more information about each property.

Note: The image to the right has been modified for the next few examples.

5. Click on the Name property. Notice that it corresponds to the name at the top of the Property Panel "Page1: Page component". The other half of this line is the kind of component that the line is referring to. The component is a Page component and thus it is recognized as that here. Type in any name in the Name property and see the change reflected in the top line. The Name property is an example of an edit box of type String.
6. Click on the Bin property down arrow. Notice that this displays a drop-down menu. Drop-down menus give many predetermined options from which to choose.
7. Click on the Grid Spacing input box. This is an example of a Numeric type property. To see what this does, change the Grid Spacing property value and look at the Page when done.
8. Click on the ellipse (…) on the Parameters property to see a dialog box appear. The dialog box is a way that a property with multiple values and options displays its choices.
9. In step 3, the option Highlight Changes was checked in the speed menu (right-click on the Property Panel). Go to the PaperSize property and choose a different paper size. Notice that not only does the paper size change, but that the property is bolded. This bolding becomes helpful in determining when properties have been changed from their default or initial values. Try changing other values such as the page width and height.
10. Also, checked in step 3, is the Exclude Name, Size, and Pos Changes option. This is a very helpful option. In most cases the name, size, and component position on a Page changes often. So, to avoid highlighting these commonly changed properties, this option excludes them from being bolded. Try unselecting this option and changing these properties.

Chapter 7
Generating Output

In this Section:
• Learn how to execute and preview Reports • Explains the options in the Printing Preferences • Learn about the toolbars and options in the Report Preview • Explains the differences in the Report Designer and the Print Preview Output Options dialog • Learn how to export reports to various formats: NDR, PRN, HTML, and PDF

Overview

After a report is created, displaying and making the report electronically mobile will become the next necessary step. There are several ways to get reports onto paper or into electronic formats. This chapter will cover the different ways to display report output and will also go through simple printing.

Executing Reports

One of the most common features used in Rave is the actual report execution. Executing a report allows the user to see the output of the report design. Execution is very similar to what many developers know as compiling and actually does involve a certain amount of internal "compiling".

There are various ways that a report can be executed. It is done either via the Project menu, using the F9 function key, or clicking on the Execute Report icon on the Project toolbar.

Using the F9 function key becomes very handy since execution becomes a very repetitive and common task performed during all phases of design.

When using any of the ways to execute a report, the user is presented with various options. The Output Options dialog shows the different options available at the time of execution.

In later sections of this chapter, each option will be explained in further detail.

Preferences Dialog

Before going through each execution option, there are some preferences that need to be covered and kept in mind before executing a report. Going to the Edit menu and selecting Preferences will display the Preferences dialog box.

The Printing tab in the Preferences dialog box allows various settings to be changed that effect report execution. Some of the options that can be set here are also reflected in the Output Options dialog box that appears when executing. In this section, each of the parameters in the Printing preferences will be explained.

In Output Options area of Printing Preferences, the Show Setup Dialog indicates whether the Output Options dialog (as shown here) is displayed when executing a report. It is sometimes convenient to disable the Output Options dialog, especially if it is during design and testing time. The drawback to this is that some of the parameters that are contained in the dialog box would have to be set before hiding this dialog. Also, any changes desired would have to be done either by re-activating the dialog box or specifying the changes in the Preferences dialog.

The default report print destination is determined by the settings under Print Destination in the Printing Preferences, and is reflected in the Output Options dialog box. Changing the value in the Print Destination area of the Printing Preferences will directly effect the default selection in the Output Options dialog box. There are two options that can be chosen: Preview and Printer. Selecting either one, by choosing the appropriate radio button, will make that option the default value and when the Output Options dialog appears the default value will be selected.

The Preview Grid Options act in much the same way as the Grid (which are the lines seen on the Page in the Visual Designer) does during design time. These Options allow a Grid to be displayed on the Print Preview screen underlying the Report. The Grid settings allow changes for the Print Preview Page, including Grid changes for color, style, and spacing adjustment. If the spacing is both 0 horizontally and vertically, no grid will be displayed in the preview. But, placing numbers in the horizontal and vertical spacing settings, will space the grid appropriately on the Print Preview Page. The Line Color will change the color of the Grid. The Line Style drop-down menu will give different options of line styles.

The Preview Options, as the name indicates, only affect previewing. There are six parameters that can be set. When using the Preview, the window displayed by default is in normal state. Like any other Window, the Preview can be Minimized or Maximized. To set the window display, use the drop down menu in the Initial Window State setting.

The Zoom Factor indicates the default zoom percentage when the preview is first displayed. The Zoom Increment specifies the Zoom percentage that will occur by each click on the Zoom In/Zoom Out icons. As with the Page Designer, the preview screen can have a horizontal ruler, vertical ruler, or both by specifying this in the Ruler Type (use the drop-down menu to make the selection). The measurements can be indicated in either centimeters or inches.

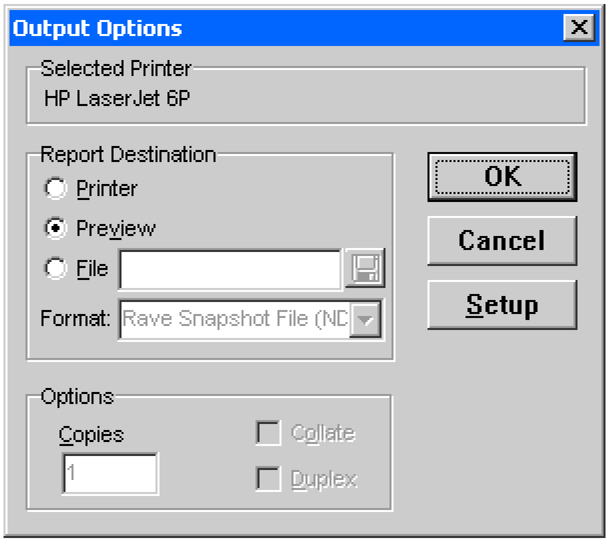
The ShadowDepth is the level of shadow that is displayed on the preview screen and it is only for on screen appearance and aesthetics.

The Monochrome Preview Display check box is there for backward-compatibility with some video cards that would give problems when displaying reports in color. This is not recommended for use, but is there if needed.

Report Preview

Although the Preview is primarily used for previewing a report, the Preview option has many more capabilities than what is intuitively understood from its name.

Begin by executing a report. When the Output Options dialog appears, the Preview radio button can be selected. Clicking OK will continue with the Preview process.



Output Options

Selected Printer

HP LaserJet 6P

Report Destination

○ Printer

● Preview

○ File

Format: Rave Snapshot File (NC

Options

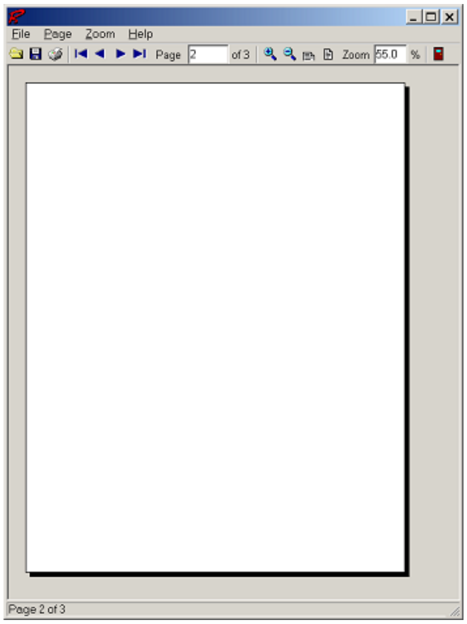
Copies

1

□ Collate

□ Duplex

The Preview screen is used to see the output of the designed report. The actual appearance on the screen resembles that of the printed report. The screen is based on a page-by-page output. There is a toolbar at the top of the Preview screen that allows navigation as well as other functions, which are explained in the following paragraphs.



Most of the functions permitted in the Preview screen can either be performed using the toolbar or through the Preview menu options. We will go through and explain the menu options in the rest of this section.

The first group of icons (separated by a vertical line on the toolbar), can be found in the File menu. They are used for opening, saving, printing, and exiting the Preview of a report.

Although the Preview screen displays the current report in the designer, it can also be used to load a previously saved report (with an NDR extension). This can be done by clicking on Open in the File menu. A dialog box appears prompting for the desired report file. When the NDR file is opened, the report that was previously in the Preview is discarded. This does not mean that the report can not be regenerated. To get the previous report, simply re-execute the report and choose Preview. Only the report that was Previewed upon execution can be regenerated. Other reports can only be revisited if they were previously saved or if their projects are re-opened and re-executed.

The Save As is used to save the report in the Preview. Click on Save As and the Save File dialog appears. Use this dialog to save the current report to various formats, including PDF and HTML (these two formats are covered later on this chapter). Another format that a report can be saved to is the NDR report type, again this is a Rave snapshot of the report.

Although reports can be printed directly by choosing Execute and then Printer, the Preview screen also provides this option, since many times it is necessary to check the preview of the report before actually printing it.

The remaining functions directly affect the viewing of the report on the Preview screen. The second section on the toolbar and the items in the Page menu are used for navigation between pages. The buttons used to designate the options will look familiar to database users, since they follow the same convention. The first button is used to move to the first page of the report. The second button moves to the previous page. The third button is used to move to the next page, and the last button can be used to move to the end of the report. On the toolbar, there is a page indicator, which at all times reflects the current page and the number of total pages. This page indicator can also be used to go to a page directly by typing the page in the edit box. Or use the Go to Page in the Page menu. For example, entering 10 in the edit box will display the 10th page of the report (providing there are 10 or more pages available).

Zooming is also available in Preview. The magnifying glass containing a plus sign will zoom in on the page, while the magnifying glass with a minus sign will zoom out. Similar to the page edit box, there is a zoom box that is used both for displaying the actual zoom and allowing the user to enter a specific percentage value. Just type in a number and the Preview will be zoomed in or out to that zoom percentage.

There are also two preset zoom values for fast navigation. The first one is the Fit to Page Width option. By clicking on this, the report can be adjusted so that the width of the page takes up the entire region of the preview screen. The second preset is Fit to Page. It can be used to view the entire page on the screen. The side effect of this is that although the report can be viewed as a whole, the actual contents might be difficult to read. Normally this can be used to get a general overview of the report layout.

The preview window can be exited by either clicking on the Door icon located on the far right of the toolbar or by clicking on the "X" button on the window (like any other window).

Executing to the Printer

Printing can be done, as mentioned before, via the Preview screen or directly when executing the Report and choosing Printer as the Report Destination. One thing to notice is that the Output Options dialogs are a little different when selected from the Rave Designer and from the Print Preview.

The Output Options dialog that appears from the Rave Designer has more options than just printing, as shown in the image.

In the Output Options Dialog there is a Setup option for Printing. The Setup button is underneath the Cancel button and can be used to adjust the printer settings prior to the actual printing. This includes options such as whether the report should be printed in landscape or portrait, the resolution of the printer, the color depth, paper size, etc.

The dialog box also includes several Options such as the number of copies to be printed, whether the copies should be collated, or duplex printed.

In the Output Options Dialog from the Report Preview (see image), there are some different options available. The Print Range has three different options. The All option means to print all pages in the report. The Selection option means to print a selection of pages not adjacent to each other, i.e. "1,3,5,7" or "1,9-5,20". The Pages option prints a range of consecutive pages, i.e. "1-10" or "15-19".

NDR & PRN Files

Printing to a file can be done several ways. We will discuss two that can be done directly from the Output Options dialog. The Output Options dialog is obtained by executing the report (use the F9 key or execute through the Project menu).

The NDR, or the Rave Format, is a Rave snapshot of the report. An NDR file can be seen and opened from the Preview window, but it cannot be changed or edited in the Rave Designer.

The PRN file is the Native Printer Output type file. When this option is selected, the information that is usually sent to the printer to print a report is instead sent to a file. This file is saved based on the file name the user gives.

To create either type of file, simply select desired format and then fill the empty File edit box with a name for the file.

The NDR can also be made using the Save As option in the File menu in the Report Preview window. Using the Save as type drop-down menu and entering a File name will create the NDR file.

There are other ways to print to a file. These are only two formats. We will cover PDF and HTML file in the next sections.

HTML

Rave has the capability to save reports as HTML files. This makes it easy to have reports readily available over the Internet or through a company intranet. HTML files can be created in the Visual Designer and in the Report Preview window.

To save a report as HTML from the Visual Designer, first get to the Output Options Dialog. Do this by using the Execute Report from the Project menu, or by using the F9 key.

In the Output Options dialog box, choose the File option from the Report Destination area. Using the Format drop down box to get to the HTML option will set the format to HTML. Clicking on the disk, will allow the user to set the path of where to save the file to and to name the HTML file.

Once a report is saved and properly formatted it may be helpful to first Preview the report. Preview the report by executing the report and choosing Preview from the Output Options dialog. After looking the report over in the Report Preview, choose the Save As option in the File menu. This will bring up the Save File dialog, where a report can be saved into HTML.

To save the HTML format, in the Save as type drop-down menu, select HTML files. Then type in the name of the new HTML file in the File name edit box. When done, click Save.

Each page of the report will be saved as an individual HTML file. For example, a three-page report would produce three pages of HTML. On each page, navigations links to the previous and next pages are provided. Once saved, the HTML files can be viewed with any Internet browser.

PDF

A truly useful report is one that is portable over many different platforms. The PDF feature of Rave makes this possible with the help of a free viewer. A report can be saved into PDF format, and through the use of the Free Adobe Acrobat Reader, any person can open, view, and print the report.

This option is available through the Execute Report (also F9) option in the Visual Designer File menu or through Save As in the Report Preview File menu. Using the Report Preview allows the user to see the layout of the report before saving it into PDF format.

After selecting Execute the Output Options dialog will appear from which the format can be chosen. Also, the file name of the PDF can be set.

In the Print Preview, using Save As will bring up the Save File dialog and from there you can save the report as a PDF type, as well as choose the file name.

To save a PDF, in the Save as type drop-down menu, select Adobe PDF. Then, type in the name of the new PDF in the File name edit box. When done, click Save.

RTF

Rave also has the capability to save reports as a RTF (Rich Text Format) file. This file format can be loaded into many word processors for a wide variety of reasons. For example, your report(s) could be included in a company document as an attachment. Or a report could be loaded into a word processor and then selected lines could be edited (highlighted, bold) to emphasis a point to a particular audience.

This option is available through the Execute Report (also F9) option in the Visual Designer File menu or through Save As in the Report Preview File menu. Using the Report Preview allows the user to see the layout of the report before saving it into RTF format.

After selecting Execute the Output Options dialog will appear from which the output format can be chosen. Also, the file name of the output file can be set.

In the Print Preview, using "Save As" will bring up the "Save File" dialog and from there you can save the report as a RTF type, as well as choose the file name.

To save a RTF, in the "Save As" type drop-down menu, select RTF. Then, type in the name of the new RTF in the File name edit box. When done, click Save.

Exercise: Changing Printing Preferences

Change Print Destination

1. Before beginning this exercise, create a new Project and place a few visual and text components on the Page.
2. First, go to the Printing Preferences tab by either using the Edit Preferences icon on the Designer Toolbar, or by selecting Preferences in the Edit menu.
3. Click on **Printing** in the Preferences option area (left of the screen).
4. Look at the Print Destination area and select **Preview**.
5. Now close the Preferences window by selecting the **OK** button.
6. Execute the report by pressing the **F9** key, or by selecting **Execute** from the **File** menu.
7. When the Output Options dialog appears, notice that the Preview option is the default value selected. Click **Cancel** when done.
8. Now, open the Printing Preferences again, and select **Printer** as the Print Destination. Click **OK** when done.
9. Execute the report again, and in the Output Options dialog, notice that the Printer is now the default value selected.

1. Before beginning this exercise, create a new Project and place a few visual and text components on the Page.
2. First, go to the Printing Preferences by either using the Edit Preferences icon on the Designer Toolbar, or by selecting **Preferences** in the Edit menu.
3. Next, make sure that Preview is selected in Print Destination.
4. Uncheck the **Show Output Options Dialog** in the Output Options area.
5. Select **OK** to close the Preferences dialog.
6. Execute the report by pressing the **F9** key, or by selecting **Execute** from the **File** menu. Notice that the Output Options Dialog did not appear; instead the execution went directly to the Preview. If Printer had been selected in the Print Destination, the report would have been sent to the printer.
7. To get the Output Options dialog back, go to the Printing Preferences and re-check the **Show Output Options Dialog** in the Output Options area.

Grid Lines and Options in Preview

1. Before beginning this exercise, create a new Project and place a few visual and text components on the Page.
2. First, go to Printing Preferences by either using the Edit Preferences icon on the Designer Toolbar, or by selecting **Preferences** in the **Edit** menu.
3. Next, make sure that **Preview** is selected in Print Destination and uncheck the **Show Output Options Dialog** in the Output Options area. We will be exploring the Preview for this exercise, so using the Print Destination and **Show Output Options Dialog** to

execute quickly to Preview will be very helpful.

4. When in the Printing Preferences dialog, make sure the initial values of the Preview Grid Options and the Preview Options are exactly as seen in the image.
5. Click **OK,** when done.
6. Execute the report and just view the Preview of the window.
7. Now close the Preview window, and Open up the Printing Preferences.
8. In the Preview Grid Options, change the **Horizontal Spacing** and **Vertical Spacing** to 1. Then change the line color by clicking on the color wheel.
9. Click **OK** when done.
10. Execute the report and notice the lines on the report in the Preview. These lines will not appear on a print out, but are there if needed to help check layout and formatting.
11. Now, close the Preview and go to the Printing Preferences again. Change the **Line Style** in the Preview Grid Options area to **Dash**. Also, change the **Initial Window State** to **Minimized**.
12. Then, in the Preview Options, change the **Ruler Type** to **BothIn**. This will display both the horizontal and vertical ruler in inches in the Preview. Also, change the **Shadow Depth** to **10**.
13. Click **OK** when done.
14. Execute the report.
15. First of all noticed that the Preview is minimized. This was set by the Initial Window State. To expand the window to see the Preview, click on the first or second button.
16. Once the window is restored or maximized. There are several things to notice. First see the dashed gridlines on the Page. These can be useful to check object alignment on the Page. Also, notice the vertical and horizontal ruler. At the very far right edge of the Page, notice the deep shadow behind the Page.
17. When done, go back into the Printing Preferences dialog. Put all settings back to their original values. The image shown displays our values. These values will be used for the remaining exercises in this chapter.

Exercise: Preview and Creating Portable Files

1. Before beginning, create a New Project. And on four pages, place text and objects.
2. Link the four pages together. Selecting the Page and the Property Panel was covered in the previous chapter. Select the first page and in the Property Panel use the drop-down menu in **GotoPage** to select **Page2**. Do this for every page. Select page 2, and make **GotoPage** equal to **Page3**. Select page 3, and make **GotoPage** equal to **Page4**.
3. Next, go to the Preview window by executing the report. Select Preview from the Output Options dialog.
4. At the top of the Preview window on the toolbar, there should be an edit box that has the number 1. That area should read Page 1 of 4.

Navigating through Preview

5. Since there is more than one page, use the navigation buttons on the toolbar or in the **Page** menu. Use the **Go to Page #** to jump to any page in the report.
6. Next, use the zoom tools to get a closer and an overall look of the report pages. This also will allow you to become more familiar with moving around in the Preview.

Creating an NDR File through the Preview

7. To create an NDR (a Rave Snapshot File) from the Preview Window, go to the **Save As** option in the **File** menu.

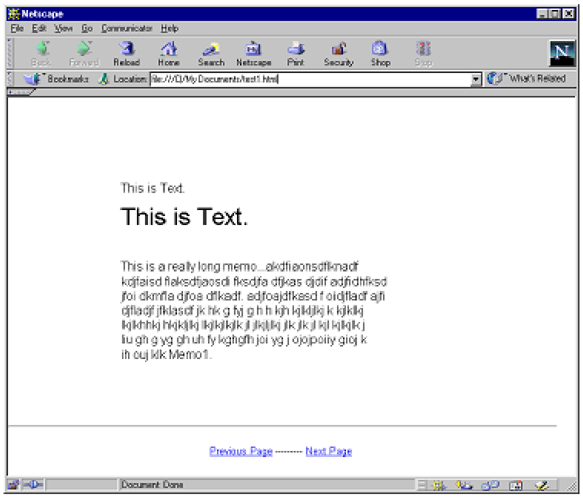
8. When the Save File dialog appears, use the **Save as type** drop-down menu to select NDR files. And put in the name of the NDR file in the **File name** edit box, in this example we used Text.ndr. Click **Save** when done.
9. Even though the file is already in the Preview, let's try opening the Test.ndr file.
10. Choose **Open** in the **File** menu.
11. The Open File dialog will appear. Look for the Test.ndr file in the directory you placed it. Click **Open** when done.
12. The Preview will most likely look the same, as we did not change the report. But, look at the top of the Preview window to where the name of the file is listed. So, as you peruse through different NDR files, the name will always be listed at the top of the Preview. Nothing in an NDR file can be changed; it is simply a way to view the report.

Creating a PDF file through the Preview

13. Since we are no longer using the executed file, close the Preview window. Execute the file and select **Preview** again.
14. Now, select **Open** from the **File** menu.
15. In the Save File dialog, select Adobe PDF from the **Save as type** drop-down menu. Then, type in a name in the **File name** edit box, this example uses test.pdf.
16. Click **Save** when done.
17. At this point you can go out to where your file was saved and open the PDF file. If you do not have the PDF reader, go to www.adobe.com and download the free reader.

Creating HTML Files

18. In the Preview window, select **Open** from the **File** menu.
19. In the Save File dialog, choose **HTML File** in the drop-down menu for **Save as type**. Then type in a name in the **File name** edit box.
20. Click **Save** when done.
21. When done you can go to the HTML file that was saved. We saved our html files in My Documents. You will notice that there are four pages of the test.html, with consecutive numbers after the test title. Each page in the report will have an associated HTML page.
22. Below is the first page of our test.html. This is test1.html, as it is the first page. Note that at the bottom there are navigation links to the get to the next pages.


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
What's Related

file:///C|/My Documents/test1.html

This is Text.

This is Text.

This is a really long memo...akdfiaonsdfknadf kdjfaisd flaksdfjaosdi fksdjfa dfjkas djdif adjfidhfksd jfoi dkmfla djfoa dfkadf. adjfoajdfkasd f oidjfladf ajfi djfladjf jfklasdf jk hk g fyj g h h kjh kjlkdjlkj k kjlklkj lkjlkhhkj hkjkljkj lkjlkjlkjlk jl jlkjljlkj jlk jlk jl kjl kjlkjlk j liu gh g yg gh uh fy kghgfh joi yg j ojojpoiiy gioj k ih ouj klk Memo1.

Previous Page -------- Next Page

Document: Done

Exercise: Printing a Report through the Preview

1. In the Preview window, select **Print** from the **File** menu.
2. This will bring up the Output Options dialog. Use any of the print ranges to get any of the pages of the report that you want.
3. When done, close the preview window.
4. Then execute the report, and select Printer. This too will send your report to the printer.

Exercise: Printing to a File (NDR & PRN)

1. Execute a report and select **File**.
2. In the **Format** drop-down box, there will be two options. **Rave Format** (.NDR) and **Native Printer Output** (.PRN).
3. Make the desired choice.
4. In the edit box next to **File**, type in the name of the file.
5. Click **OK** when done

Chapter 8

Utility Toolbars

In this Section:

- Alignment Toolbar introduced with Order, Spacing, and Alignment of components
- Designer Toolbar introduced and its relation with Preferences explained
- How to read the Color Palette Toolbar and its designations
- Font Editor briefly introduced and explained
- Use of the Line Editor Toolbar is explained
- Functions of the Project Toolbar are introduced
- Zoom Toolbar functions covered

Tools are Tools

Even though the toolbars are called toolbars, there are two distinct types, relating to the function performed. There are Utility Toolbars and Component Toolbars.

A tool is typically used to modify an item already on the Page, usually a component. A component is an object placed on the Page, like a Region, Shape, Line, or Barcode (for more information regarding components, see the Components Overview chapter).

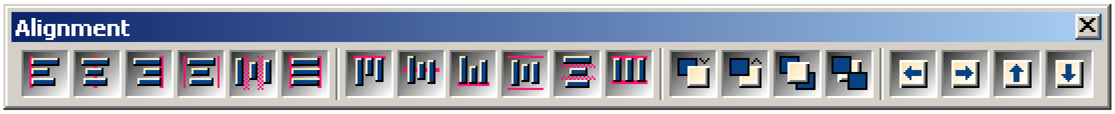
This chapter will cover the Utility Toolbars, which contains tools (Color, Alignment, etc.) used to modify components. Utility Toolbars include the Alignment Toolbar, the Color Palette Toolbar, the Fill Toolbar, the Font Editor Toolbar, the Line Editor Toolbar, the Project Toolbar, and the Zoom Toolbar.

Using Tools

The Utility Toolbars act on components and the Page Designer. Either a component or the Page has to be selected prior to using the toolbar. For example, to change the fill color of a Square, select the Square and then click on the required color.

By right clicking on a component, a popup menu will appear giving access to other options that can change the component. These options are also available from the toolbars; the popup menu just gives faster, easier access.

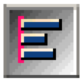
Alignment Toolbar



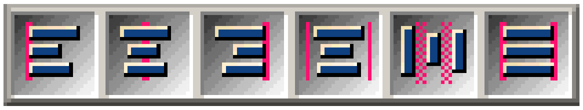
The Alignment Toolbar is used to align, order, and move a group of components on the Page. It is important to note that some of the Alignment options only make sense when more than one component is selected.

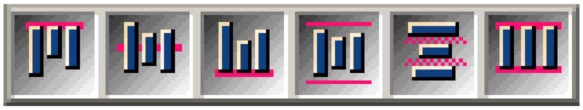
The first step is to select the components that need to be aligned. To select multiple items, hold the shift key down and click the mouse on each object that needs to be aligned.

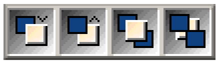
For most alignment options, the ***first*** component selected is what determines the alignment position. For example, when choosing Align Left, all selected components will be left aligned to the position of the first selected component. A red line on the icon designates the actual alignment.



The Alignment Toolbar is divided into four sections by separators. The first two sections control horizontal and vertical alignment options. Both allow components to align on the edges, centers, or to be spaced equally.

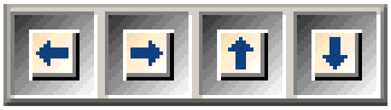




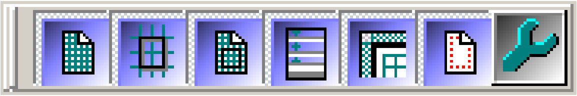


The Order Alignment determines the Z order (depth along the Z axis) between components. Order Alignment does not require more than one component to be selected. To establish the Z order of a component, select the component and right-click the mouse button to display the popup menu. From the Order submenu choose one of the four options: Move Forward, Move Behind, Bring to Front or Send to Back. The figure to the right shows a Rectangle with an Ellipse imposed in front of it. To move the Ellipse to the back of the Rectangle, right-click the Ellipse (by right-clicking on a component previously unselected, it will select it and bring up the popup menu) and choose Send to Back from the Order submenu.

There are an additional four buttons on the Alignment Toolbar, Tap Tools, which allows small incremented movements of a component (or group of components). To move selected component(s), click on the corresponding arrow to move the component(s) towards the desired direction.



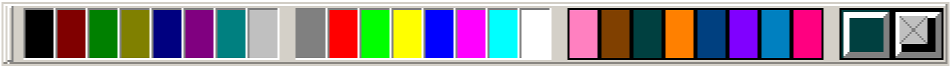
Designer Toolbar



The Designer Toolbar is used to control the Preferences for the Page. All of the Designer Toolbar items toggle. By clicking on a button, the property of the component associated with the button will change status. To change it back to the previous state, click on the button once more. The top toolbar shows what the buttons look like when they are not pressed (toggle status off) and the bottom toolbar is the resulting toolbar when the buttons have been depressed (toggle status on).

"Activate Grid"	controls whether the grid lines are visible or not on the Page Layout
"Snap To Grid"	controls whether the objects on the Page 'snap'. Snap means that the component will only move and resize according to where the Grid Lines are present
"Grid On Top"	determines whether the Grid Lines are always on top of the components or not (i.e. are not hidden when components are placed on the Page Designer)
"Always Show Band Headers"	controls the displaying of band headers
"Show Rulers"	toggles the appearance of guide rulers in the environment
"Show Waste Area"	displays the space between the Page boundary and the end of the Page Setup Boundary
"Edit Preferences"	will bring up the Preferences dialog

Color Palette



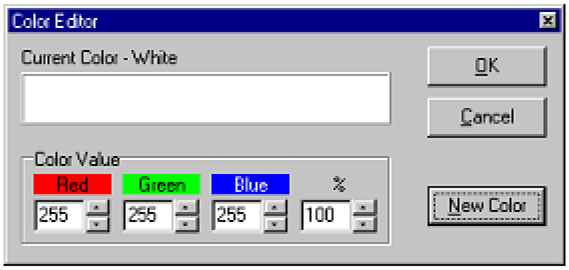
The Color Palette is the tool used to change a color of a selected object such as a Line, Text component, or the fill area in one of the shapes (Circle, Ellipse, Rectangle, or Square). Color palettes are composed of several sections that show custom, standard and current colors. The left button on the right side displays the currently selected Primary, while the right button shows current Secondary color. All other color squares serve as the Color Palette from which you can select.

To change the color of an object, select the object and either left or right-click on the Color Palette. A left-click selects the Foreground Color (Primary) while the right-click selects the Background Color (Secondary).

When a Primary Color is selected, a "1" will be placed on the corresponding color indicator on the palette. Similarly, a "2" will be placed indicating the Secondary Color.

To change the color of an object (like a Rectangle), first select it and then click on the desired color with the left mouse button. To change the Fill Color, click on the desired color with the right mouse button.

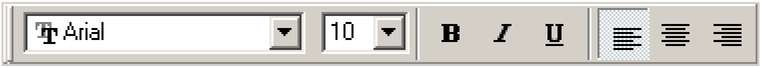
Double-clicking on the user defined Custom Colors (right side) or on the Foreground and Background Color boxes will open a more detailed color control window called the Color Editor. From the Color Editor, custom colors can be created by adjusting the red, green and blue values, as well as the saturation of the color by increasing or decreasing the color percentage.



To create and save a new color, use one of the Custom Colors to enter into the Color Editor, then click on the New Color button and choose the desired color from the Color dialog box. By entering the Color Editor from one of the Custom Color boxes, the new color will be saved. If the Color Editor is entered from either the Primary or Secondary Color boxes, the color will only be saved in those boxes until a new color is chosen (either by clicking on the right or left mouse button or by the creation of another color).

It is important to note that the actual exhibited colors are dependent on the display settings of the computer. If there are issues with colored objects when transferring reports from one system to another, check the display color settings. A 256-color setting only uses 256 colors for display, where as a True Color setting uses millions of colors. Thus, for example, if a Report is created on a system set at True Color, and then viewed on another system set at 256 Colors, some objects may appear grainy or to have the wrong color shadings.

Font Editor



The Font Editor can be used to change the type, size, attributes and alignment of any text component in the report. Its aspect and functioning is very similar to Font Toolbars found in the most commonly used Word Processors.

To change the Font settings of a Text component, first select the component. The font type can be adjusted by choosing a font from the drop-down list. Similarly the size can be changed by clicking on one of the available sizes for the font in question. The **Bold**, *Italic* and Underline buttons are toggled depending on the state of the text. To set the font to **Bold**, just click on the Text component and then press the **Bold** button. The button will then remain depressed until it is clicked on again or until focus is moved to a different text object that does not have the **Bold** set. The *Italics* and Underline button work in much the same way. There are three alignment buttons that indicate the alignment of the text within the area it is contained, and the options are left, centered, and right-justified. All text-based components have a property called Font that appears in the Property Panel when the component is selected. Any changes made to the Font, are reflected both in the font toolbar and in the Font property.

NOTE: It is important to note that if a font is chosen on the development machine and this font is not available on the deployment or any other machine, Rave will assign the next most similar font to the one chosen.

Line Editor

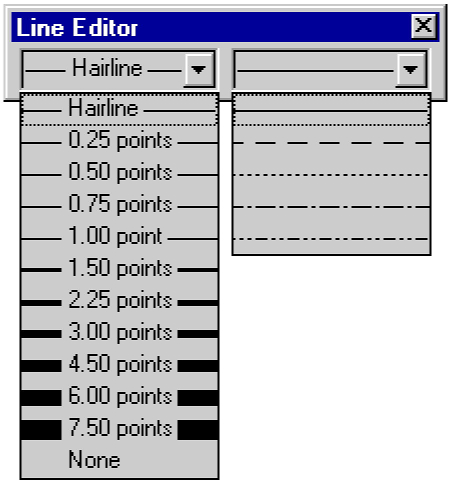
Use the Line Editor to modify the style and thickness of a line or the border of a shape. The Line

Editor Toolbar has several selections to choose from including line styles (solid or combinations of dots and dashes).

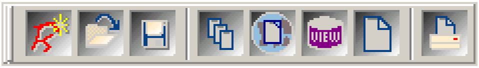
To change the style of a line, first select the line or shape, and then choose the point size of the line as Hairline. Finally, choose the style of the line.

To change the thickness of a solid line, first select the line or shape. Then choose the point size desired.

Line style only (dashes, dots, etc) applies if the line thickness is hairline. If a line is set to a certain style (like dashes) and then given a thickness other than hairline, the line style will be changed to a solid line.



Project Toolbar



The Project Toolbar provides the basic functionalities for a Project. It gives a quick easy way to complete the common tasks associated with managing a Project. All options available on the Project Toolbar are accessible through the main Project menu in Rave.

The first three icons deal with managing projects:

	"New Project"	creates a new Project
	"Open Project"	opens a previ ously saved Project
	"Save Project"	saves the current Project

The next three icons deal with creating specific aspects of the current project:

"New Report" will create a new Report that will be contained in the same Project file (.rav file). Rave can have more than one Report per Project file

"New Global Page" creates a new Global Page

"New Data View" allows the creation of DataViews that are associated to the current Project. Please see Connecting to Data chapter more information about DataViews.

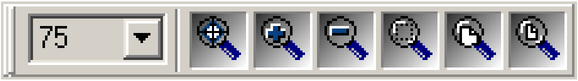
The last two icons deal with the currently selected Report.

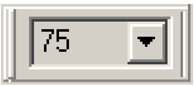
"New Page" creates a new Page in current Report

"Execute Report" execute (or print) the current Report

Zoom Toolbar

The Zoom Toolbar allows the user to Zoom In to view the details of a document better or Zoom Out to view the overall Page uniformity. Within the Zoom Toolbar, there are several ways to zoom into and out from the details of the document.





To Zoom by Percentage, the Zoom Factor drop menu has many preset percentage factors to Zoom in and Zoom out. Simply choose the desired Zoom Percentage. If one of the presets is not adequate the you can specify a percentage by typing the desired percentage into the edit box next to the zoom factor drop menu.

Another method to get a customized size is to click on the Zoom Tool (magnifying glass). Drag the tool over the specified area that needs to be magnified.

For a simple manual Zoom In and Zoom Out, click on the magnifying glasses with the "+" and "-" sign. Nothing will need to be selected to use this tool. Each click is set to a specified incremental amount that is set in the Preference settings. To get to the settings area, go to Preferences in the Edit Menu. In the Edit Menu, select the Designer Tab. In the area called "Zoom Increment", increase or decrease the percentage by using the up/down arrows to increase the percentage or simply type in the percentage.

To zoom in on selected items, use the magnifying glass with the dashed box. First select the desired item(s), and then click on this tool. The viewable area will frame all item(s) selected.

.

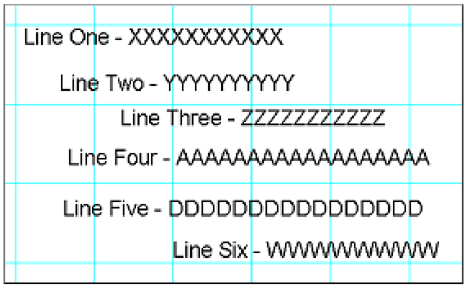
Use the "Page Width" button to zoom to the full width of the currently selected Page.

Use the "Whole Page" button to Zoom to the full page size of the current page.

Exercise: Aligning Components

This exercise will demonstrate how to align components. The example images have six numbered lines of text.

1. To begin, drop several Text components onto a Page. In this example, the components have been number labeled



Line One - XXXXXXXXXXX

Line Two - YYYYYYYYYY

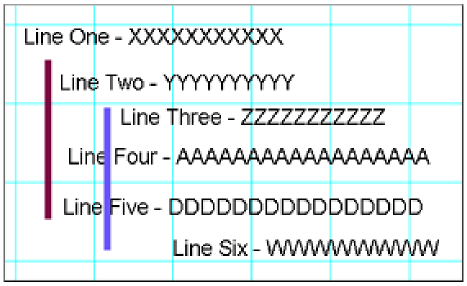
Line Three - ZZZZZZZZZZZ

Line Four - AAAAAAAAAAAAAAAAAA

Line Five - DDDDDDDDDDDDDDDD

Line Six - WWWWWWWWWWW

2. Lines two, four, and five will be aligned. Lines three and six will also be aligned



Line One - XXXXXXXXXXX

Line Two - YYYYYYYYYY

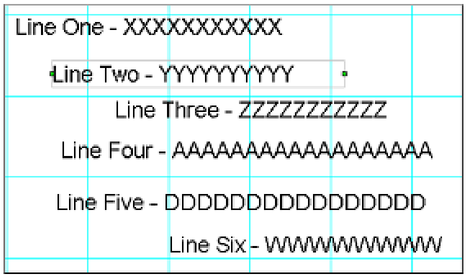
Line Three - ZZZZZZZZZZZ

Line Four - AAAAAAAAAAAAAAAAAA

Line Five - DDDDDDDDDDDDDDDD

Line Six - WWWWWWWWWWW

3. Select Line Two. This will be the primary component that the rest of the components alignments will be based on



Line One - XXXXXXXXXXX

Line Two - YYYYYYYYYY

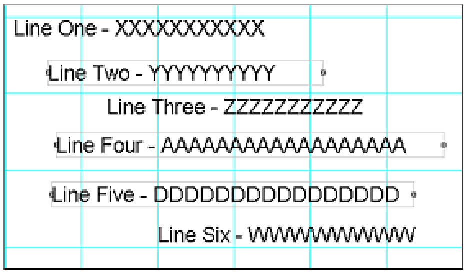
Line Three - ZZZZZZZZZZZ

Line Four - AAAAAAAAAAAAAAAAAA

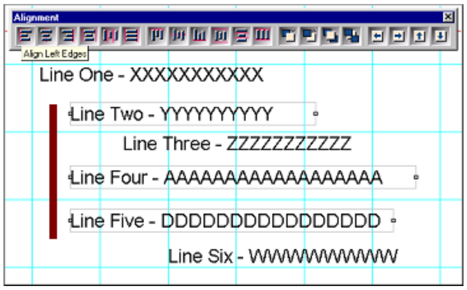
Line Five - DDDDDDDDDDDDDDDD

Line Six - WWWWWWWWWWW

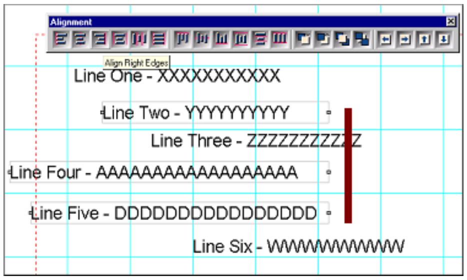
4. While holding the shift key, select lines Four and Five to be aligned with Line Two. Notice the light gray border that appears with pips



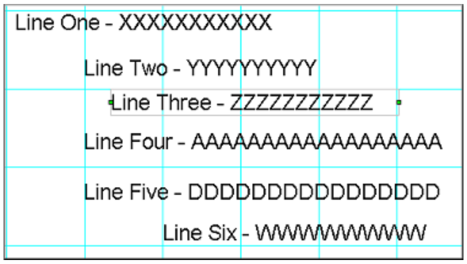
5. Once all components have been selected, choose Align Left Edges on the Alignment Toolbar. Notice that all three components left edges have been aligned



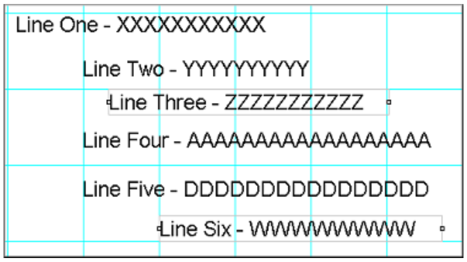
6. To further show how the alignment works, click on the Align Right Edges button on the Alignment Toolbar



7. Go through the rest of the Alignment tools to see how they function
8. Let's align the second set of text. First, select the primary component to base the alignment on, in this case Line Three



9. Then select the other component(s) that will be aligned, by holding the shift key and clicking on Line Six



Line One - XXXXXXXXXXX

Line Two - YYYYYYYYYY

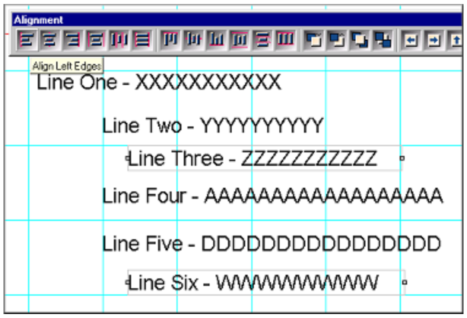
Line Three - ZZZZZZZZZZZ

Line Four - AAAAAAAAAAAAAAAAAA

Line Five - DDDDDDDDDDDDDDDD

Line Six - WWWWWWWWWWW

10. Then click on the Align Left Edges to align the components.



Line One - XXXXXXXXXXX

Line Two - YYYYYYYYYY

Line Three - ZZZZZZZZZZZ

Line Four - AAAAAAAAAAAAAAAAAA

Line Five - DDDDDDDDDDDDDDDD

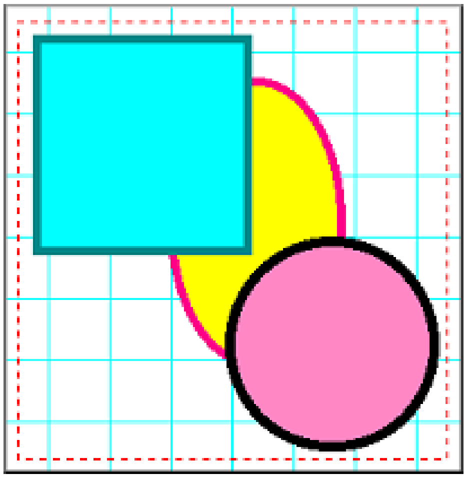
Line Six - WWWWWWWWWWW

Exercise: Ordering Components

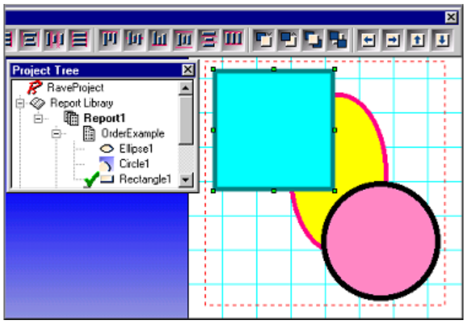
Ordering refers to the ordering of components along the Z-axis. This is the axis that comes out of the Page.

To order components along the Z-Axis, the order buttons on the Alignment Toolbar must be used.

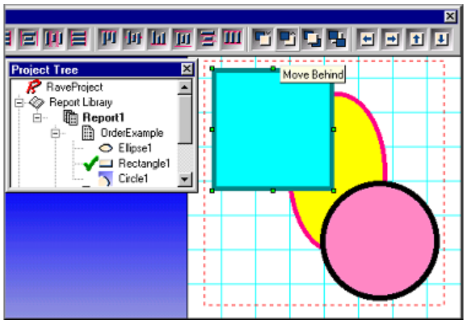
1. To begin, there must first be components on page. These can be any components. For this example we will use a Rectangle, Oval, and Circle



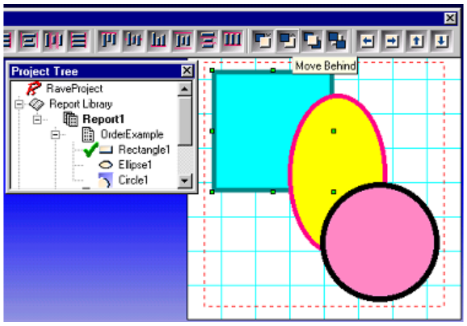
2. First select a component to move. In this example, the rectangle will be moved behind the Oval. Notice in the Project Tree the placement of the Rectangle component



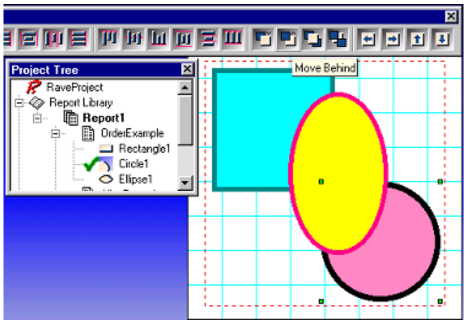
3. Once the rectangle is selected click on the Move Behind button on the Alignment Toolbar. Notice that after clicking the button once the rectangle has not moved behind anything yet, but on the Project Tree the rectangle has moved up the tree



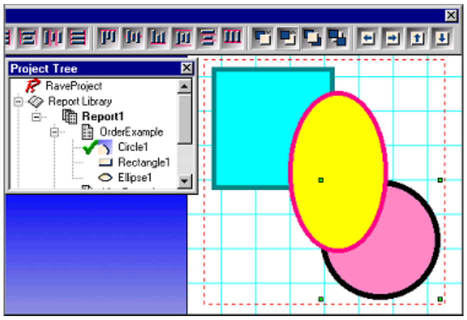
4. Try clicking on the Move Behind button one more time. The rectangle will move up the Project Tree and the Rectangle on the page will move behind the Ellipse



5. For the second component, select the circle and click the Move Behind button. The Circle component will move up the tree and move behind the Ellipse



6. Click on the Move Behind button once more and the Circle will move up the tree one more time and remain behind the Ellipse



Exercise: Snapping to the grid

Snapping to Grid refers to the moving and resizing of components according to where the grid lines on the page are present.

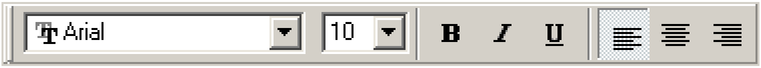
1. To begin, bring up the Preferences Dialog box by selecting Preferences from the Edit menu
2. Select the Defaults tab and look for the area labeled Grid Spacing
3. In Grid Spacing, change to .02 units. Then look for the area called Draw Grid Every
4. In Draw Grid Every, enter 1 line (type or use the up and down arrow). Draw Grid Every 1 (that's what we chose) line, will show every Grid line made. If this were changed to Draw Grid Every 3 lines, then for every third line the Grid would be marked in the Visual Designer
5. Next go to the Project menu and choose New. This will create a new Project, as well as a new Report Page, that is based on the preferences that were just changed in the previous steps. Remember the Grid lines serve only as page guides for designing reports; they do not show up in any report executions
6. Go to the Designer Toolbar and click on the Snap To Grid
7. Now that the Report Page is created, drop a rectangle on the page
8. Now move the rectangle around and notice that the rectangle only moves according to where the lines are located. Try using the tap buttons. Notice that the component doesn't seem to move until the tap button moves are on one of the grid lines
9. Now to go the Designer Toolbar and click on the Snap To Grid button to turn the feature off
10. Move the rectangle and notice that the rectangle will move to positions between the grid lines

Exercise: Changing Line Size and Color of a Rectangle

1. Create a new Report Page
2. Drop a rectangle component on the Page
3. Look at the Color Toolbar. Notice the color blocks marked 1 and 2, you can review Primary and Secondary Designations in this chapter
4. To make the color change more notable in this example we will increase the boundary lines of the Rectangle. Bring up the Lines Toolbar. Select the Rectangle. Then in the first Lines Toolbar drop-down box, choose a larger border size. In this example, the borderline has been increased from hairline to 2.25 points.
5. Left-click on any color in the Color Toolbar. In this example the color Olive was selected. Notice that the border has changed to Olive. Also notice that the Primary Color has changed. The Primary Color is the color of the border.
6. **Right-click** on any color. In this example the color Blue was selected. Notice that the fill inside the Rectangle has changed to Blue. Also notice that the Secondary Color has changed. The Secondary Color is the color of the inside of the Rectangle

Exercise: Changing Fonts

1. Create a new Report Page.
2. Drop four Text components on the Page.
3. Select a component, and then view the Font Toolbar.



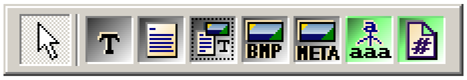
4. On drop-down menu displaying the Font Name, click on the down arrow and choose a desired font.
5. Go to the next component and repeat Steps 3 and 4. Do this for all remaining Text components.
6. Also, try using the font-sizing drop-down menu. As well as the **Bold**, *Italic*, and Underline options
7. The last three buttons to the far right deal with placing the font as left-justified, centered, and right-justified. Practice using these options to get the desired formatting effect.

Chapter 9
Standard Components

In this Section:

- Explains the difference between a Text component and a Memo component
- How to use a Section and its relationship to the Project Tree
- How to insert BMP and META files

Overview



The Standard Toolbar contains seven components: Text, Memo, Section, Bitmap, Metafile, FontMaster, and PageNumInit. Several of the Standard Components are used most frequently when designing reports. This chapter will provide a detailed explanation of the first five. The remaining two (FontMaster and PageNumInit) are covered in the Advanced Components chapter.

Access to the Standard Toolbar is achieved in the same way as with other toolbars. If it is not present on the screen, it can be displayed by selecting it from the Toolbar submenu in the Tools menu.

Text

The Text component is useful for displaying a single line of text on the report. It acts basically like a label that can contain simple text (not data). When placed on the report, the Text box is surrounded with a box that indicates its boundaries. This can only be seen while the text component is selected (represented by the pips).

The component can be used for labels, such as for figure or graphical titles, floating text, form titles, and in general just about anything that is denoted by a single line of text.

When the component is selected, the length of the text can be adjusted by resizing it using the pips. Note however that the height of the text is self-adjusting and does not require further intervention by the user.

Like any other text-based component, there is a Font property that can be used to change the font type, size and style. The color can be set using the Color property. The actual text of the component, which is seen on the screen, is denoted by the Text property.

There is an additional property called Rotation, which can be used to rotate the text by a specified number of degrees. The effect can only be seen at runtime.

Memo

Memo components are similar to Text components. However the most noticeable difference is that Memos can contain multiple lines of text. Memos can be used on forms for areas of explanation, and for titles or comments that are longer than one line.

Similar to the Text component, the border around the Memo can only be seen when the component is selected. When setting the Font of the Memo, all the lines of text in the Memo will have the same font. *It is not possible to set different fonts for certain parts of the text*.

To change the text in the Memo, go to the Property Panel and use the Text property. The Text property will show "(Memo)" and an ellipse button, which are three dots. Click on the ellipse button. This will open up the Memo Editor. Text can be entered into the Memo Editor.

One of the main problems with using multi-line text components is that the text might overlap in height on to other components on the Report. To prevent this, set the ExpandParent property to **True**. This will expand the Parent of the Memo component to properly accommodate the Memo component.

Once the text is entered, the Memo box can be resized. The text within the Memo box will be repositioned accordingly. If there appears to be text missing from the Memo box, after it has been entered into the Memo Editor, resize the box to allow all text to be visible.

Section

The Section component is used to group components. This provides advantages, such as allowing all the components that form part of the Section to be moved together with one mouse-click, as opposed to moving each component individually or trying to select all components before moving.

To use the Section Component, first press the Section Component button and then use the cursor/arrow to make a selection on the Page. When the selection is completed, there will be a box marked by dashes with green pips. Within this area, place any components that need to be grouped together.

The Project Tree becomes very helpful when dealing with the Section component. From an expanded Tee, it is easy to see what components are in each Section. Each Section and component can also be easily selected simply by clicking on the appropriate object in the Tree.

If an object is ever placed beyond the viewable section area, the Project Tree can be used to select the object. Once the object is selected, the object's border pips can be seen and thus its location. The object can then be deleted or moved (hint: to move use the tap buttons in the Alignment toolbar).

As seen from the relationship with the Project Tree and the Section component, there is a need to understand Parent-Child relationships between components. To read more on these types of relation ships see the Project Tree Panel chapter, it contains a section on Parent-Child Relationships. The Section component also becomes very important when mirroring. Mirroring will be covered in the Managing Report Projects chapter. For now, just know that once the Section component is mastered, using this with mirroring will become extremely helpful.

Bitmap and Meta File

The Bitmap and Metafile components enable images to be placed in a report. Bitmap supports image files with the extension ".bmp", and MetaFile supports image files with the extensions ".wmf" and ".emf".

To insert an image, first click on either the BMP or the META buttons on the Standard components toolbar, then click any location on the Page. A placeholder box with an X will appear on the Page, and the box may be bigger than what the image will be. This can be fixed after the image is loaded.

Once the BMP or META placeholder is on the Page, go to the Property Panel and to the FileLink property. Click on the ellipse, the button with three dots.

This will bring up the Open Dialog. Find the image file, select the file, and click Open.

Once the image is loaded, resize the placeholder pips on the image border.

Exercise: Text vs Memo

The decision to use either a Text or a Memo component is quite easy. If there is only a single line of text to be placed on a Page, use the Text component. Use the Memo component when there is more than one line of text to be used.

To place and change a text component:

1. Click on the Text component located on the Standard Toolbar. The component button will change colors to designate activation.
2. Click once on the Page. The Text component will appear on the Page. Also notice that a new component will be added to the Project Tree under the Page that it was placed.
3. Go to the Property Panel and in the Name property change the name of the Text component.
4. In the Text property, change the default text to the desired text to be displayed on the Page.
5. View the Property Panel and the Page. Notice that the Project Tree has the Text component changed to the name entered in the Name property. Also, on the Page the component has the desired text typed into the Text property.

To place and change a memo component:

1. Click on the Memo Component located on the Standard Toolbar. The component button will change colors to designate activation.
2. Click once on the Page. The Memo component will appear on the Page. Also notice a new Memo component will be added to the Project Tree under the Page that it was placed.
3. Go to the Property Panel and in the Name property change the name of the Memo component. This will change the name of the Memo component in the Project Tree under the Page that the component was placed on.
4. In the Property Panel, go to the Text property. Click on the ellipse, the button with three dots (…).
5. The Memo Editor will appear. In the Memo Editor, type in the text that needs to be placed on the Page. The Memo will recognize returns made with the Enter key. Remember though, that the Memo will be subject to the size of the Memo box on the Page.
6. When the text has been entered, click OK.
7. When done, the Memo component can be moved or resized by using the green pips.

Exercise: Section

To create and use a Section Component:

1. Click on the Section component. The component button will change colors to designate activation.
2. Click on the Page, and a dotted box will appear with pips at its boundaries. This is the Section component.
3. In the Project Tree, notice that the Section component was added under the Page it was

placed.

4. While the Section component is selected, place any other component into the Section. When done, notice that the placed components will now be under the section in the Project Tree.
5. Once components are placed into a Section, all the components can be moved together. To try this out, place the cursor on the border of the Section, click and hold, then drag the Section around the Page.
6. The Section can also be resized. Click and hold any of the pips and then move the cursor in the desired direction, the Section box will resize.
7. When done, click anywhere on the Page to deselect the Section component. The Section is an invisible border displayed by a dashed rectangle, but it will not be printed.

Exercise: Placing and Resizing Bitmaps

1. To complete this exercise, find a Bitmap (BMP) image on your computer. The image will have a ".bmp" extension at the end of the file name. Remember the location of this image.

2. Create a New Report Page. Do this by selecting New Report Page from the Project menu.

3. BMP On the Standard Toolbar, click on the BMP button.

4. Then click once on the Page to drop the Bitmap component. The green pips will appear around the edges to indicate that it is selected.
5. While the Bitmap component is still selected, go to the Property Panel. Look for a property called FileLink Property. There will be three dots, called an ellipse button, to the right of it. Click on the ellipse button.
6. Using the Look in drop-down menu and selecting the image in the box below. Once The Open Bitmap dialog will appear. Find your Bitmap image on your computer by the file is selected, click Open.
7. The image will appear on the Page with green pips around the image.
8. Now look at the Property Panel and look at the property MatchSide. This property deals with the proportions of the image when it is resized using the pips.
9. Click on each of the MatchSide choices and move the image around a bit to see the effects of the choices.
10. This is msBoth. The pips on all sides allow the resizing of all pictures in all directions.
11. This is msHeight. This helps in resizing the pips according to height, while still keeping the image proportional. Using the top and bottom pips will resize the image.
12. This is msInside. This allows proportional resizing, but all resizing is done with in the limits of the pip. The image doesn't leave the borders of the pips.
13. This is msWidth. This does proportional resizing according to the left and right pips of the image.

Chapter 10

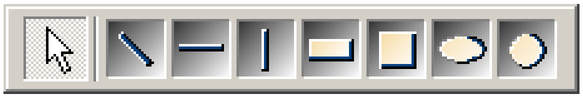
Drawing Components

In this Section:

- Explains how to place a Drawing Component.
- Describes all the Drawing Components.
- Discusses Pixels vs. Points.

Drawing Component Basics

All the Rave graphical components are created using the Drawing Components.



To use any of the Drawing Components, click on the desired component in the Drawing Toolbar then click once on the Page or other container component. This will create the object on the Page or other container component. Once the component object is created it can be resized to the desired dimensions by dragging the pips around the selected object border. Also, to color and stylize the components, use the Fills (where applicable), Lines, and Colors Toolbars. For more information on these toolbars, see the Utility Toolbars chapter.

The Property Panels of each drawing components are almost identical. The Name property may be the only thing that distinguishes one Property Panel from another.

The Line component can be used to draw a line in any direction. Angled lines may be used to draw lines between rotated lines of text. They may also be used as diagonal lines inside a square giving the effect of a box with an "X" in it.

Line styles may be set for the line; however, you should be aware that when setting line styles it is best to stick with a line width of one. Line styles are often ignored when the line width is any greater than this.

The HLine component may be used to draw horizontal lines. One of the common uses for the horizontal line is to drop one into a DataBand and set it so that it is positioned at the very bottom of the Band. It is also set to stretch the whole length of the Band. When the Band is printed, the line creates the effect of having the Band information printed on lined paper.

The VLine component may be used to draw vertical lines. One of the more common uses for the vertical line is to drop one into a DataBand and set it so that it is positioned in between various data fields that are being printed. The height of the line is set to the same height of the Band. When the Band is printed, the vertical line gives the effect of having the information printed in nice neat columns.

The Rectangle component may be used to draw rectangles. One of the more common uses for the Rectangle is to drop one into a DataBand and set it so that it is sized to completely fill the band. It is then moved to the back so that other components dropped on the Band will print over the top of the rectangle. When the Report is printed, this gives the effect of having a nice block around the rows of printed data.

The Square component may be used to draw squares. A common usage of this component is showing checked or unchecked boxes. The checked boxes are created with the use of the Square component and the Line component, or perhaps by simply by placing an "X" in the Square. This is useful when trying to illustrate items that have been selected.

The Ellipse component can be used to draw an ellipse or oval shape, more simply defined as a flattened circle. If you want to change the shape of the Ellipse, you can do this by clicking on the Ellipse with the left mouse button. Then move the mouse over one of the pips that will appear. Press the left mouse button again, and while continuing to hold the mouse button down, move the mouse until the Ellipse is the desired size and shape. You can also resize the Ellipse by setting the properties directly using the Property Panel. Simply enter new values for the Width or Height properties.

The Circle component can be used to draw a circle with a constant radius. If you want to change the size of the Circle, you can do this by clicking on the Circle with the left mouse button. Then move the mouse over one of the pips that will appear. Press the left mouse button again, and while continuing to hold the mouse button down, move the mouse until the Circle is the desired size. You can also resize the Circle by setting the properties directly using the property panel. Simply enter new values for the Width or Height properties.

Pixels vs Points

For every Drawing Component, the Property Panel has two properties that deal with Line Width. The LineWidth property is a numeric value for the width of the line, which depends on the LineWidthType. The LineWidthType property actually lets the user determine which type of value the line will be set. The Types available are Points and Pixels.

When creating drawing objects, it is important to determine in what format the audience will be seeing the Report. This is important when determining if Points or Pixels will be used for the line width type.

Points are a unit of print, not a unit of screen space. Points are the unit of measurement used most commonly with paper design. It has meaning in the paper medium. Unfortunately, points are meaningless on the screen and the web. Due to platform and resolution differences, something like 14pt can mean many things. What it does not mean is a specific unit of screen size. Pixels, on the other hand, are defined unit. A pixel is always equal to a pixel.

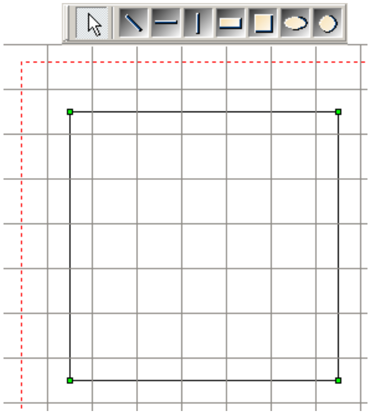
Points and Pixels Preferences are dependent upon what the designer wishes the user to experience. To let the user adjust the text size at will or let the user's internet browser preference determine the size, specify points. If the design depends on a rigid text size or if the text size must be consistent across platforms, specify Pixels.

To print the thinnest possible line, use a Line Width of 1 pixel. For thicker lines, values in points should be used for consistency across printers with different resolutions.

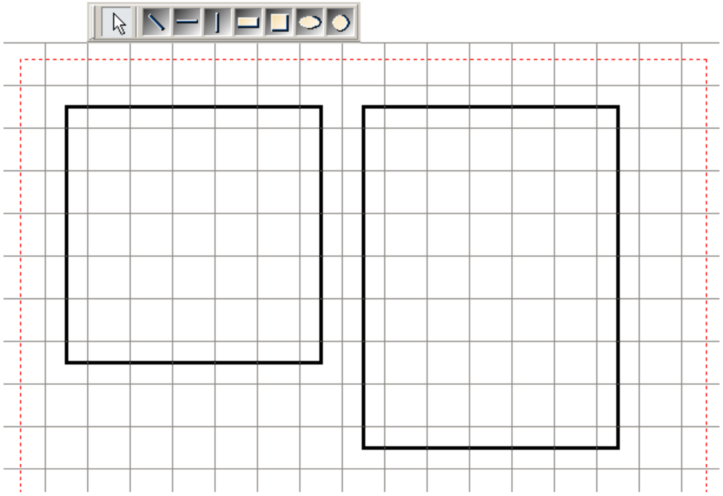
Exercise: Creating Drawing Components

1. First create a new Page. Use New Report Page from the File menu, or use the icon on the Project Toolbar.
2. Now, go to the Drawing Toolbar. Pull it down from the Tab area, so that we can concentrate on it here.
3. Click on the Square, notice that it's button changes to a depressed button and changes color. Click your cursor on the Page and hold down the left mouse

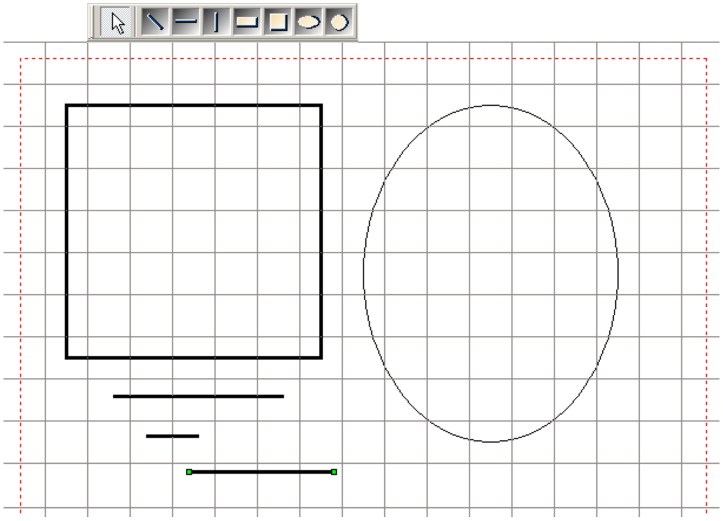
button while dragging out the shape of the square.



4. Next, click on the Ellipse. Click on the Page and hold down the left mouse button while dragging out the shape of the Ellipse. The Ellipse outline will look like a Rectangle, but once the mouse button is let go the ellipse shape will appear.

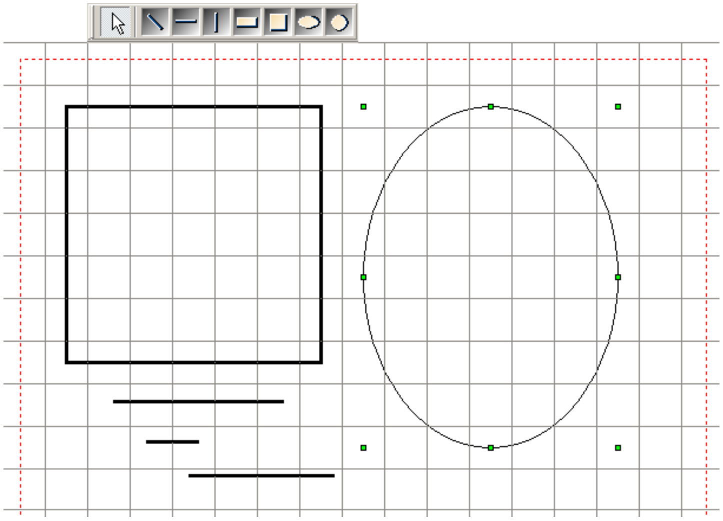


5. Then, click on the Horizontal Line and draw them on the Page. Do this for several lines. Make them different lengths.



6. One thing to notice for every shape, when the mouse button is released

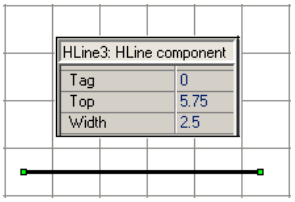
pips outline the shapes. When all shapes are selected the pips appear.



Exercise: Alignment

1. First create a new Page. Use New Report Page from the File menu, or use the icon on the Project Toolbar.
2. From the Drawing Toolbar, click on the horizontal line button then click on the Page. The line will appear as the default line size on the Page. The clicking on the button and clicking on the Page is what we call dropping an object on the Page.

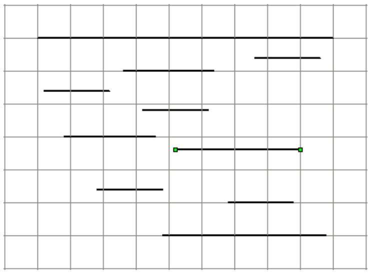
3. Select the horizontal line by clicking on it.



HLine3: HLine component

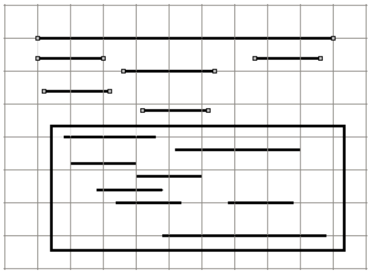
Tag	0
Top	5.75
Width	2.5

4. Go to the Property Panel and change the default value of the line by typing 2.5 into the Width property edit box.
5. Drop down at least 10 additional horizontal lines about an inch below the first one.

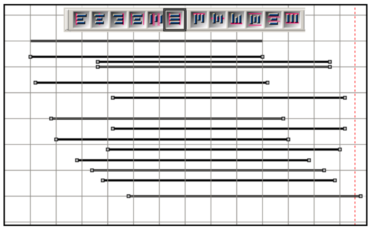


6. Drop down one more horizontal line about 3 inches below the first group of lines.
7. Select the top horizontal line by clicking on it. The green pips will appear at the ends of the line when selected.
8. While the first line is still selected, hold down the Shift

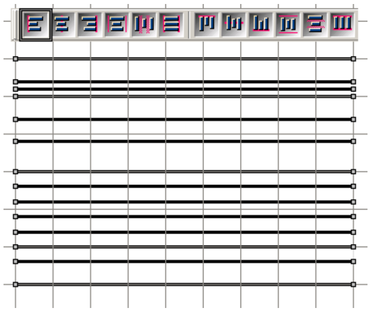
key and start clicking on the other lines. This will add all the other horizontal lines to the selection. To select the remaining lines faster, while the first line is selected and any other additional lines are also selected, keep holding the Shift key and click on the Page and drag the resulting box over the rest of the lines.



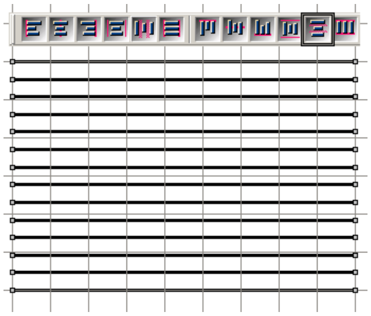
9. In the Alignment toolbar, click on Equate Widths alignment button. This will make all the lines the same width as the first selected line.



10. On the Alignment Toolbar, click on Align Left Edges button. This will align all the lines to the left edge of the first selected line.



11. Next, on the Alignment toolbar, click on Space Equally Vertically. This will space the lines equally away from each other on the vertical axis. For more exercises on Alignment, see the Utility Toolbars. Chapter.



Chapter 11

Database 101

In this Section:

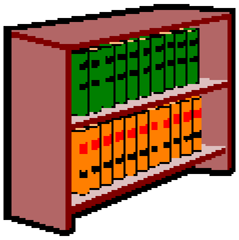
- Gives an overview of databases.
- Explains the needs to make up a database.
- Describes the data table structure and how it relates to reporting.
- Explains database relationships.

Overview

Information to the left, information to the right, information above and below, we are swimming in a sea of information. The advent of the computer age has not helped. In fact, we now have the ability to store absolutely huge amounts of data. This can easily result in information overload for both individuals and managers. Something is needed to manage this overload potential. The first thing is that the information must be organized and stored in an efficient manner that aids in the quest to extract answer(s) from that wealth of data. One of the methods to retrieve answers is to use a reporting tool (like RAVE) to organize and display the answers in a manner that is easily understood by the reader of the report. But before a reporting tool can extract the information, it must be aware of the structure of the source of information. That container of the information in simple terms is a data table.

This section is intended to assist those that are not familiar with database design goals or might need a quick refresher. This section is not designed to cover this subject in depth, as this is best left to the many thick books already published on this subject and to more formal educational courses.

What is a database?



Let's begin with some real world examples to describe the concepts of a data table. For example, a library contains lots of information stored in rows and rows of shelves containing books, pamphlets, etc., which is usually arranged according to the Dewey decimal system. To find a book, a card catalog is used to look up the subject or author, noting the Dewey decimal code. Then, the desired book is found and retrieved by going through the stack that contains the book code (if it isn't checked out).

Look up options are far more flexible if the library has a computerized card catalog. Now picture a library with no Dewey decimal system, no order but still the same amount of books. Try and find something in that library. It would be very tough and take a long period of time.

Now, change the words. Change book to information and library to database and we are on the way to explaining what and why of database management system. A library is a collection of knowledge (books) arranged by the Dewey decimal system.

A data table is a collection of information arranged according to some predefined table structure. We will now examine what is needed to design or use a data table.

Terms

To better understand what it takes to create a database, it is necessary to understand the database lingo.

Database Management System

A Database Management System (DBMS) is the complete system for managing data (information). There are many commercial examples of companies that are known for their DBMS programs. Some of the more well known include Informix, Interbase, Microsoft Access, Microsoft SQL, Oracle, Sybase, and many others. A DBMS can be confused with the physical data file(s). It must be noted that the data table is only one part of a DBMS. However, the real world does require that when extracting information from a DBMS that the structural concepts of a data table be understood. Fortunately, most of the terms are common between the various DBMS. This does not mean a person needs to be an expert in the database field, just knowledgeable.

Table

A data "Table" is a collection of things like phone numbers, names and addresses. The "report" of this example table would be a telephone book. Of course, a telephone book has some restrictions, like do not include unlisted phone numbers in any print outputs. In some cases, the phone company will have a billing address that is different from the address location for the phone itself.

Now look at a phone book and notice that it is arranged in rows and columns. A row (sometimes called a record) shows the items related to a single member (person) of that table. Each column (sometimes called a field) is giving a specific piece of information about that member, like their name or phone number. While we are here, the order of the phone book (table) is done with an index (sometimes called a key). So, a phone book could be arranged by last name or by phone number.

Structure

Each specific piece of information about a member of a table has attributes that define it for its DBMS.

Typical attributes for each table (file) include:

- date the table was created
- date the table was last modified
- number of columns defined for table

Typical attributes for each row (record) include:

- Length
- Record number
- Status (locked - deleted)

Typical attributes for each column (field) include:

- Name - a column name that usually "describes" its contents

- Type - character, numeric, date, time, blob etc.
- Length - maximum size of this item (might include number of decimals for numeric types)

A data table is comprised of information arranged according to a pre-defined structure. The structure of each table is important, because the row and column definitions are used by a reporting tool to extract, arrange and show information.

In the phone table example, there are the following fields: name, address, and phone number information. How are those columns (fields) defined? They are defined by the name of the column; meaning name might be called "name", address called "address", and phone number called "phone". But defining the main fields may not end at these simple definitions. For example the "name" field might be broken into three or more sub-fields, like "first", "middle" and "last". These three pieces as well as the first three main fields could then be manipulated as needed by any reporting tools. The three name sub-fields could be combined for a "full name" output or the first letter of the "first" name could be combined with "last" to make a short version of the name. Also, "phone" could be called "number" or "telephone" and the phone number could probably contain only the numbers with no formatting characters. Rave makes the task of understanding the table structure relatively easy as it does most of the work. Once the data connection is made, Rave will extract the table structure and from there the columns (fields) can be chosen as desired.

Query

Rarely is there a need to print all the information in a database. Frequently, there is a need to reduce data by selecting a range of rows (records) in a data table. Data reduction is an important part of information management. One method of accomplishing a reduction is to do a query of the table to only show those rows that meet the range limits that need to be retrieved.

One of the more common query methods is called SQL, which stands for Structured Query Language. Some data tables are SQL compliant. A SQL query allows inquires of a DBMS for information, and the DBMS searches its tables according to the given SQL statement and return a "set" of rows (records) that meet that query.

Client-Server

A method of setting up computer programs used by many people, where the database resides on a central computer accessible to all (the back end), and the user interface resides on the user's computer (the front end). All selection of specific data rows and processing is done back on the host computer minimizing the transmission of data through the network cabling. Significant improvements are achieved to the overall system performance by only transmitting relevant data through the cables. Thus, a well-configured Client-Server system can yield excellent performance gains in applications used by several people.

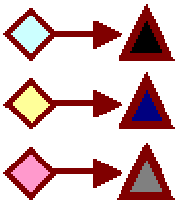
Relational Table

Now the phone book example is good for simple data tables (flat files). However, it is not an efficient design for many real-world needs. We can stay with the phone company, just not the phone book for a more typical need. Every month the phone company needs to send a bill out to each customer. This bill is a good example of a "master-detail" type report. One part of the bill comprises the master (static) information about the customer like: their name, billing address and phone number. Another part of the bill contains the detail (variable) information, like an output line for each phone call made including its length and cost. The variable information could be only one page or for some customers many pages.

There is one important detail in any master-detail work, understanding the concept of a primary key (link key). When a customer gets a phone bill, that customer expects only to receive a listing of their calls; they don't expect calls made by other people to be listed. In order for this customer bill to be created, this means that there needs to be only one master row, called a primary key, for all the detail rows. In the customer bill, the primary (link) key would be the phone number. The primary key must be unique, which in general usually makes it more difficult to change a primary key once it has been assigned. But, there are some DBMS that allow the primary key to be a combination of columns (fields).

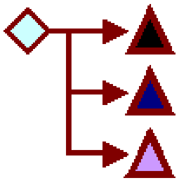
That is the quick review of Database operations. To ensure efficient design, with table structure control, there are a few more subjects to be aware of like: "Client-Server", "Normalization", "Relational DBMS", "One to One" "One to Many" and "Many to Many" relations.

One-to-One



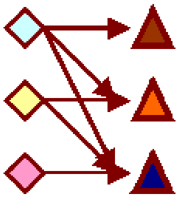
One-to-one is two separate tables with the same primary key. Therefore, knowing the primary key allows look up of data for a first row in one table, and for a second row in another table. One to one relationships are rare because it's usually easier to put both tables' columns into a single table. One-to-one relationships probably don't meet the first level of "Normalization" rules.

One-to-Many



This is accomplished by placing the primary key value of the ONE side of the relationship into an ordinary column of the MANY side of the relationship. For instance, if each employee can belong to only one department, then you could have a "department" column in the employee table, which, for each employee, would be filled with the primary key value for the row of his department in the department table.

Many-to-Many



This is difficult to explain, as the real-world examples are complex. If there were several employees that were part of several departments, then this would be a many to many situation

View

Some DBMS have an ability for a user to define a form to show contents of information taken from one or more other tables. A view is a definition of a form and has no data of its own. It often includes a query that is executed whenever the view is the subject of a command. Use views to control access to individual columns in a table, or to make two or more tables appear as one.

Reporting

How does all this relate to designing reports? With the definitions defined, understanding the report design structure and options will be easier. Now you are ready to read the sections on DataView Connection, DataField which will tell how to use Rave to connect to your data table(s) and how to identify the specific column (field) contents to the Rave designer.

Chapter 12
Connecting to Data

In this Section:

- Explains what a database connection is.
- Show the different methods of retrieving data from a database.
- Explains the SQL Editor and how it is used.

Database Connections

A reporting tool would be quite pointless if it did not allow methods of displaying information contained in databases. One could only go so far with designing non-data reports. Connecting to data in Rave is one of the more powerful features, not only because of the wide variety of databases and methods supported, but also due to the simplicity of the steps required to make a report data-enabled.

A database connection in Rave is the primary pillar for connecting to a database with SQL DataViews.

Creating a Database Connection

To create a new Database Connection, click on the DataView icon on the Report toolbar, bringing up the Data Connection Wizard. The first step is to select Database Connection from the list of options available. When done, click Next.

After clicking on the Next button, the second step of the wizard displays the different type of connection options available. The selection depends on the different Rave DIBL Links that are installed on the system. DIBL stands for Database Independent Layer. It is a proprietary system that allows interactions with databases independently of what these database system are (whether it be SQL Server, Interbase, Oracle, etc).

In a folder under the Rave root directory, there should be a folder called DIBLLinks. Inside the folder there are a certain amount of files with extension rvd. The files are loaded when the application first starts. Depending on the number of files, the number of available connection options varies.

It is possible to have many types of connections displayed in the Database Connection Type Dialog screen. It is left up to the user to decide which connection type is the most appropriate. Fewer connection types may be displayed depending on which drivers were installed.

Once one has been selected, the corresponding dialog box will appear in the next step asking for the connection details. This varies from one to another, but generally consists of a path to the database, a server name if the database is not local and optionally a username and password to access the server.

After creating a database connection, it will appear in the Project Tree from where it can be accessed. The properties for a Database Connection can be shown just like any other component by clicking on the connection component in the Project Tree panel.

There are several properties that are specific to the Database component. The most important ones are the AuthDesign and AuthRun properties. Many times, the platform (and server) that one develops the report on does not coincide with that of deployment. Specifically, one characteristic that is most likely to change is the access codes to the server (username and password). Rave has been designed keeping this in mind, and once again making the deployment task easier. For this, the AuthDesign and AuthRun properties can be used to provide design-time and runtime (deployment) information regarding the database and server.

AuthDesign contains the parameters that were specified when the Database component was made using the wizard and it can be changed by clicking on the ellipse button in the property field. AuthRun is the information for deployment.

LinkType represents the type that was selected when creating the Database connection. Again, similar to other components, there are common properties such as Name, FullName and Description, etc, which are not be explained again to prevent redundancy.

Direct DataViews (BE only)

DirectDataViews provide a link to data connection components located within an application. When a DirectDataView is created, a list of all available data connection components is displayed. The *Name* property value of the data connection components provides the link between it and the DirectDataView (through the *ConnectionName* property in Rave). If connecting to data connections that have event code attached to them, you must have the application actually running and the form containing the data connection components created.

Driver DataViews

DriverDataViews perform the same function as a DirectDataViews except they are self contained and typically obtain their data set from a SQL statement. DriverDataViews also require the use of a valid database connection.

Once the database connection has been established, click on New Data Object to create a new DriverDataView.

The Wizard will take the user through the necessary steps to complete the configuration. After selecting DriverDataView from the dialog box, choose the database connection that it is going to be connected to. If the connection to the database is successful, the wizard will then display the Query Editor dialog box. Before continuing with an explanation of the DriverDataView, let's examine the Query Editor.

Query Property Editor

The Query Property Editor is used for generating a SQL statement that returns a result set from the database.

There are two ways to use the editor. The first is to take advantage of the graphical interface for constructing the SQL sentence. On the left side of the dialog box, a list of tables available in the current database is displayed. By simply dragging and dropping from the list to the gray area, the corresponding SQL statement will be generated.

Once the table object appears, by clicking on the tick boxes, the list of fields that should appear in the result set can be defined. By default "*" is marked, which means that all fields will be returned. To select individual fields use the tick fields to make the appropriate selections.

Tables can also be linked (joined). To do this, simply drag another table over and trace a line from one field in the first table to another field in the second time (the figure above shows a join between PRODUCTID of table BUGS and PRODUCTID of table PRODUCTCODES. To view the results from here, click on the Results tab and the data will be displayed in a grid.

Alternatively, the editor can be used to write an SQL sentence directly. To do this, click on the Editor button that appears at the bottom. To enter a user-defined SQL statement, click on the "User Defined SQL" checkbox and type in the desired SQL.

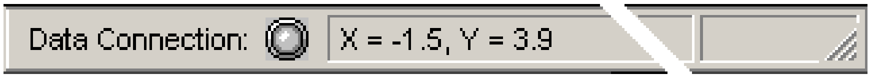
Once the query has been constructed, clicking on the Ok button will return control to the Rave designer.

The SQL Data View component should now appear in the Project Tree. By expanding the + sign next to it, a list of all the fields that have been selected in the SQL statement will drop down. Properties for each field can then be accessed from the Property Panel prior to selecting the field in the Project Tree.

Although fields can be placed in a report by using the corresponding component from the Report tab, it is much easier to work directly with the Project Tree to accomplish the same text. Pressing down the CTRL button, fields can be dragged and dropped directly from the Project Tree on to the form designer.

Once the DirectDataView has been setup using the Query Editor, none of the other properties are required for accessing the data.

Status Bar



Data Connection:

X = -1.5, Y = 3.9

At the bottom of the RAVE screen is the status bar. The status bar will provide some information about the item currently selected. So, watch the bar when designing a report.

The data connection LED will provi de status of the Rave data system by the color of the LED.

LED	Color	LED Status
Gray	●	Connection(s) are inactive
Yellow	●	Connection active but waiting for response
Green	●	Connection active and busy
Red	●	Connection active but has exceeded time-out delay

The X and Y figures are the coordinates of the mouse pointer. Move the mouse around and watch the X, Y amounts change. When a shape is dropped on the page, the size of a shape will appear while it is being created as seen by the dX and dY position. The "d" stands for delta.

Chapter 13

Wizards

In this Section:
• List guides to help in simple report structure. • Guides will provide detailed steps to complete a reporting task.

Wizards

Wizards are a new feature to Rave that allow certain types of reports to be created by answering a series of questions. They can be found under the "Tool" menu option. The Wizards can be added and tailored to each user's needs. It is a perfect way to minimize the end-users interface with reporting tasks.

There are two Wizards, a "Simple Table" and a "Master-Detail". Wizards can be designed to ask for the data connection and allow the user to choose the fields that are needed on a report. It is important to note that an active DataView should be available prior to running the wizards, whether it is a Direct Data View or an SQL Data View. Simple Reports are generally used for listings. Common uses include Client reports, Telephone lists, etc. The Master-Detail Wizard is used when more complex reporting is required, such as invoices, product order lists, etc.

Independent of the one that is executed, there are several steps that are common to both. These include the DataView selection, field selection, etc and are covered in greater detail in the exercises included at the end of this chapter.

It is very important that the user understands that these are general purpose reporting wizards and as such, some aspects are not treated with great detail (amount of text that would fit on a page, layout, etc). They can be used as the building blocks for a more complex report or can be adjusted to suite a particular layout required. If the intention is to build very complex reports, the wizards are not recommended for this purpose.

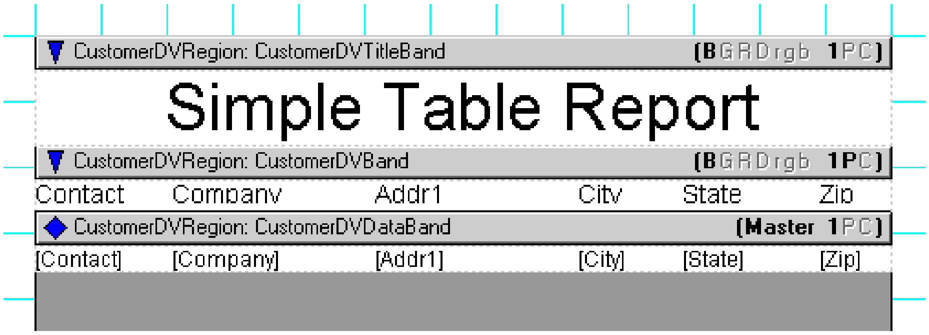
Exercises

Below are example exercises of both Simple and Master-Detail wizards. Since most of the steps are similar for both wizards, they will only be explained in detail in the first exercise. It is highly recommended to start with the Simple Wizard first to get a good grasp on the concepts presented.

Exercise: Simple Wizard

1. The first step is to select the desired DataView that will provide the data to the report.
2. Once the DataView has been selected, a list of fields that should appear on the report can be chosen. Fields are selected by clicking on the box on the left. All fields can be selected by clicking on the All button at the top. Note however that because this is a simple wizard, choosing more fields that would fit on the page will make them overlap. Manual intervention is required after the Wizard is complete to correct this problem.
3. If more than one field was selected in the previous step, the Wizard will ask for the ordering of the fields. Moving a field up will place it on the left side of the page. Move it to the lowest position will move the field to the right margin.
4. After the fields are placed in your desired order, the Report layout can be set. Values include the title and the page margins.
5. The last step of the Simple Report Wizard is to choose the fonts that are going to be used in the report. It gives the possibility of changing three fonts: title (of the report),

caption (headers of the field names) and the body (actual field values).



Exercise: Master Detail Wizard

The Master-Detail Report Wizard has a few additional steps more than the Simple Wizard Report. This is due to the fact that more than one table is going to be taking part in the report. In particular, there will be a master (for example customer information) and a detail (items ordered).

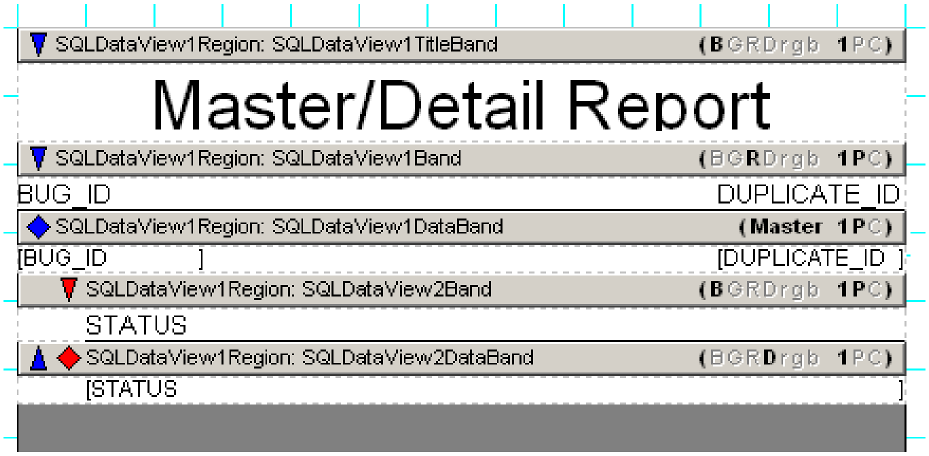
1. Similar to the Simple Report Wizard, the first step is to choose the DataView. However, in this case the DataView chosen corresponds to the Master table.
2. In step two, the Detail table can be selected. Note how the DataView selected in step 1 is no longer available to avoid errors.

Similar to the Simple Wizard, the next steps are for selecting the fields and ordering of both the master and the detail table.

A new step is determining the key fields. These are field that relate one table to the other.

After this, the remaining steps are identical to that of the Simple Report Wizard.

Once everything is complete, clicking on Generate will produce a simple master-detail report that, again, can later be adjusted and "touched up" to ones particular needs. The figure below shows a sample output of the Wizard.


SQLDataView1Region: SQLDataView1TitleBand
(BGRDrgb 1PC)
Master/Detail Report
SQLDataView1Region: SQLDataView1Band
(BGRDrgb 1PC)
BUG_ID
DUPLICATE_ID
SQLDataView1Region: SQLDataView1DataBand
(Master 1PC)
[BUG_ID]
[DUPLICATE_ID]
SQLDataView1Region: SQLDataView2Band
(BGRDrgb 1PC)
STATUS
SQLDataView1Region: SQLDataView2DataBand
(BGRDrgb 1PC)
[STATUS]

Chapter 14

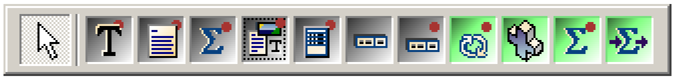
Report Components

In this Section:

- Gives more detail on using report components.
- Examples are given to aid in comprehension.

Overview

The Report Toolbar is one of the most used when working with data-aware reports. It is very vital to understand the function of each component, especially since the basics of all database reports are based on the Report Toolbar.



The following components are from the Report Toolbar that are frequently used in designing your reports:

Band, CalcText, DataBand, DataMemo, DataText, Region

The following components from the Report Toolbar will be covered in the Advanced Components section of the manual.

CalcController, CalcOp, CalcTotal, DataCycle, DataMirrorSection

The following Report components are non-visual components

CalcController, CalcOp, CalcTotal, DataCycle

There are some general characteristics of the components that need to be discussed before describing each component individually. First, let's talk about the visual color of the components. For the most part the components look similar in color to all other components, they are gray. But, at the end of the toolbar there are four green colored components. The green indicates that these are non-visual components. When placed on the Page, they cannot be seen on the screen, in preview, or in print. Instead, to 'see' the components the user must use the Project Tree to select, move, and delete the component from the Page.

Most of the Report Components can simply be placed on the Page. The only two exceptions are the Band and the DataBand. These must be placed in a Region component. The behavior of the component is also controlled by the settings made with the "Band Style Editor", which is found through the BandStyle property for both components.

A Report can be designed into a typical report, like one that uses strictly Bands. But, the unique feature of Rave is that the report can also be designed to other specifications. The Bands of the report can be moved and resized to any desired shape.

When designing a Report, the designer window can be changed to the preferences of the Report Designer.

There are several ways to control the "look" of the Bands while in the design mode. The first is whether ALL Band Headers need to be seen or not. This setting can be changed with the Designer Toolbar. Another method is to toggle the visibility of the Band contents. If some of the Band designs are stable and they need to be temporarily hidden, then try setting the

DesignerHide property to True. Only the Band(s) with DesignerHide set to False will show. This might be needed if there are a large number of Bands or a Band that occupies a large space. The setting of the DesignerHide property has NO effect on what will be printed, only what is shown on the designer Page.

Region

Before one can use a Region component properly, it is important to understand what it is. A Region is a container for Bands. In its simplest form the Region could be the whole Page. This would probably be True for reports that are a list type.

Many master-detail reports could be made to fit a single Region design. However, do not be limited by thinking of Regions as the whole Page. The properties for a Region basically deal with its size and location on the Page. Creative use of Regions will give more flexibility when trying to design complex reports. Multiple Regions can be placed on a single Page. They could be side-by-side, one above the other or stagger about the Page. Do NOT confuse a Region with a section. Region components contain Bands and only Bands. A Section component can contain any group of components, including other Region components.

When working with Bands, there is a simple rule that must be followed: Bands must be in a Region. Notice that the number of Regions on a Page is not limited nor is the number of Bands within a Region. As long as the report can be "visualized" mentally, a combination of Regions and Bands can be used to solve any difficulties faced when actually putting the visual thoughts into design. There are two band types: Band and DataBand. The latter is used when iteration is required, for example, in a master-detail report. The first is used for non-iterating needs.

DataBand

The DataBand component is a data-aware Band is used to display iterating information from a data view. In general, a DataBand will contain several DataText components.

A typical use for the DataBand would be on an invoice. An invoice normally consists of a header including information such as the date, invoice number, clients name and address, and one or more lines containing the items that are being invoiced. In this scenario, the customer table would be the master table and the items would be the detail table. Information about the items would be placed in the DataBand and the controller would be the master table.

Band

The Band component is for items that are "fixed" and do not change on the Page. In general, the Band component will contain Text and CalcText components. The primary examples of this would be headers and footers. The Band component can contain data-aware components, so a table field can be in the Band. A group footer might have a '{CustomerDV.CustomerName} Totals' on this Band.

An important property for the Band component is the "ControllerBand". This property determines which DataBand this Band belongs to (or is controlled by). When the controlling Band has been set notice that the graphic symbol on the Band will point in the direction of that controlling Band and that the color of the symbols will match.

There is a preference setting called "Always Show Band Headers". This setting will change the appearance of the Bands while in the design mode. Having this setting "off" will give an appearance closer to the actual output, but will not show the Band Headers with their symbols and codes. When first starting to use Rave it might be beneficial to try it with this setting "on" and take advantage of the visual clues provided by the Band Headers. Once comfortable with the use of Bands, change this setting to fit any necessary needs or preferences. The letter codes shown on the right of the Band are explained later in this chapter under the section called "Editor - Band Style". Basically, they give information about Band behavior. The bold letters are ON or active while the subdued letters are off or inactive.

Editor - Band Style

Go to a Band Style property on a Band or DataBand component and click on the ellipse button to open the editor for Band Styles.

This provides a simple method to select the features wanted for that Band by using the check boxes to activate or deactivate them. Note that a Band can have several different features active at a time. This means that it is even possible for a Band to be both a Body Header and Body Footer at the same time.

The display area in the Band Style Editor has been designed to represent the flow of a Report in pseudo layout style. DataBand(s) are duplicated 3 times on purpose to show that this is a repeating data area. The current Band that is being edited will be displayed with both **Bold** and Underline formatting.

Both symbols and letters are used on the Band Style Editor display area and one the Bands in the Page Layout Area and are designed be informative about each Band's behavior. The major difference between these two representations is that the Band style editor display will arrange the Bands in a pseudo flow according to the definitions of each Band. The Region display of the Bands will be arranged in the order that they are placed during design. The order of operation is controlled in some cases by this order. Headers (capital letters, BGR) will print first, then the DataBand, then the Footers (lower case letters, bgr) for each level. However, if there is more than one header defined for a particular level, then each header Band will be processed in the order that they are arranged in the Region. So, it is technically possible to put all the Headers at the top, all the DataBands in middle and all the Footers at the bottom of a Region for all levels of a master-detail. Or each level could be "grouped", with the appropriate Headers, Footers and DataBands all together for each level. Rave allows the Region layout maybe be used in a way that makes the most sense to the design flow. Just remember the order of precedence of like Bands at the same level is controlled by their order within the Region.

There are several symbols that are designed to show the Parent - Child/Master - Detail relations of the various Bands. The triangle symbol (up/down arrows) indicates that the Band is controlled by a Master Band with the same color (level) and can be found in the direction of the arrow. The Diamond symbol represents a Master or Controlling Band. These symbols are both color coded and indented to represent the level of Master - Detail flow. Remember that we could have Master - Detail - Detail where both details are both controlled by the same Master or one of the Details could be controlled by the other Detail.

The title bar of each Band contains information about that Band. On the left side of the Band is a name that indicates the Region it is in - "RegionName:BandName". The right side of Band uses several letters to remind you of the Band style settings for that Band. The order of these letters on a master Band is "MASTER 1PC". The order of these letters on a controlled Band is "BGRDrgb1PC". If the letter is subdued (gray) then that setting is inactive (off). If the letter is bold then the setting is active (ON). The following table shows the various letters and what they mean.

DataText

The DataText component is data-aware. This means it can be used to display a field from a dataset anywhere on the Page layout. For example, this could be used to print the customer information inside of a DataBand. The DataText component however is not limited to printing only database data. Through the Data Text Editor (accessible through the DataField property), Project Parameters can be printed, as can Report Variables as well as the DataFields. See the topic Editor-DataText, for more information. The LookupDataView, LookupDisplay and LookupField properties define a lookup definition to be displayed instead of the DataView:DataField properties.

Editor - DataText

There are two options available for entering data in a DataField property. The first is to select a single field using the drop option. This is fine for normal database reporting needs where only a single data field for each DataText item may be needed. However, there are many reporting requirements where various fields will need to be combined together. Two common examples are City State and Zip Code or Firstname Lastname combinations. In code this would be accomplished using a statement like the following:

City + '_' + State + '__' + Zip or FirstName + "_" + LastName

NOTE: The underscore character above represents a space for example purposes only.

The DataFhield property has a DataText Editor which assists in building complex composite fields. To do this click on the ellipsis and open the DataText Editor. This editor will give the power to concatenate fields, parameters, or variables together to build a very complex data aware text field simply by dropping the different list boxes and selecting the item desired.

There are a lot of combinations in this editor, they will be covered quickly here, but try the different combinations in practice and it should help the learning curve.

Note that the dialog box is divided into 4 groups: Data Fields, Report Variables, Project Parameters, Post Initialize Variables and Data Text. Data Text is the result window. Watch this window when inserting different items. The two buttons on the right side of this window are +, "plus", or &, "ampersand". The "plus" sign will add the two items together with no spaces while the "ampersand" will concatenate them with a single space. So, the first step is to decide on doing a + or &, then selecting the text from one of the three groups above the Data Text window.

So as an example, to add the field "OrderNo" to the "CustNo", click once on the "+" sign, go up to the DataField group, drop the DataField list box, and select "OrderNo". Then, click once on the "Insert Field" button and that will be added for the DataText window. The result in the DataText window would be "CustNo + OrderNumber". Even more DataFields can still be added. Notice the "Selected" item in the DataView group. If there is more than one data view active, then select another Data View, and then pick a field from that Data View.

However, do not be restricted to thinking of combining only DataFields. "Report Variables" and "Project Parameters" can be combined as well. Go to the "Report Variables" group and drop the list box for variables and take note of the ones that are available.

Another item available is Project Parameters. This could be a "UserName", "ReportTitle" or "UserOption" parameter initialized by the application. To create the list of "Project Parameters", select the Project node in the Project Tree (very top item). Then in the "Properties" panel there will be a "Parameters" property. Click on the ellipsis button to get a typical strings editor where you can enter the different parameters that will pass to Rave from the application, like "UserName" etc.

Caution:
Remember to use a "+" or "&" between each item that you are combining in the Data Text window. You can type in the Data Text window, so you can correct errors by highlighting, deleting or replacing erroneous entries made in the Data Text stream.

DataMemo

The DataMemo component is data-aware. This means it can display a memo field from a DataView just about anywhere on the Page layout. The main difference between the DataMemo component and the DataText component is that the DataMemo component is for printing text that may take more than one line and will thus need to be wrapped. For example, this could be used to print the remarks about a customer invoice at the bottom of each Page of an invoice.

One use of the DataMemo component is to do mail merge functions. The easiest way to accomplish this is to set the DataView and DataField property control to the source of the Memo field. Then launch the mail merge editor by clicking on the ellipsis button of the MailMergeItems property. This will allow the items to be set within that Memo that will be changed.

Note: The "Case sensitive" check box is empty (off). If case is important, then be sure to check this box.

To use the Mail Merge Editor, click on the "Add" button. Now type in the "Search Token" window the item that is in the Memo and will be replaced. After the token is entered, either type the replacement string in the "Replacement" window or click on the "Edit" button and it will start the DataText Editor that will help in selecting different DataViews and Fields.

CalcText

The CalcText component is data-aware. The main difference between the DataText component and the CalcText component is that the CalcText component is specially designed to do some form of calculation and display the results of that calculation. The CalcType property determines the type of calculation being performed and includes Average, Count, Maximum, Minimum, and Sum. For example, this can be used to print the Totals of an invoice at the top of each page of an invoice.

The CountBlanks property determines whether blank field values are included in the Average and Count calculation methods. If RunningTotal is True then the calculation will not be reset to 0 each time it is printed.

Chapter 15

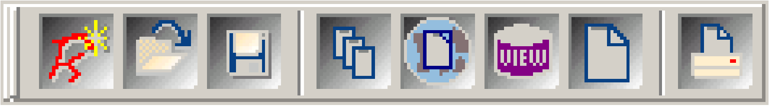
Project Components

In this Section:

- Main function of the Project Toolbar discussed.
- Project Manager and special properties covered.
- Report, Page and Global Page components and their special properties covered.
- Data Connections and Objects used to connect to Reporting Server discussed.
- Differences of Security components explained.
- Explanation of SQL and Direct DataView are given.

Overview

The Project Toolbar provides the basic functionalities for a project. These functions are essential building blocks for all reports. What makes these functions so important is the fact that these functions also provide basic building block components.



The main functions we will be concerned with the first button (Project Manager) and the second section in the toolbar (New Report, New Global Page, New DataView and New Page). These functions are used to add the supporting structure to your report design.

Each will be discussed in the following sections.

Project Manager

The Project Manager is the component from which all other components are created. Everything is placed under the Project Manager (a.k.a. RaveProject) in the Project Tree. This component is created when any new Rave file is created. There can only be one Project Manager per Rave file, this is unlike most of the other components.

For clarification, Project Manager, RaveProject, and New Project all refer to the same thing. They refer to the main Project file that contains everything.

Like every component in Rave, the Project Manager has properties. To see them, select RaveProject in the Project Tree, and then look in the Property Panel.

To quickly create a new project, simply click on the New Project button in the Project Toolbar. This will create a new file, as well as the new Project Manager.

Noted Project Manager Properties

The AdminPassword property allows the Rave Server administrator to limit who has access to the data from which the reports are created. This is important to help limit capabilities in Rave.

Parameters property allows items calculated by applications (such as Delphi) to be passed into the report to be used by other components with in the report. When clicking on the Parameters property ellipse button, the DataText Editor will allow a parameter name for each line.

PIVars Parameter is for content that is not typical, and needs to be dynamically modified within the report. This allows for content that needs to be modified in a way that is not typical of table calculation or manipulation.

SecurityControl

Report

The Report component contains the Pages of a Report. There can be multiple Reports in one Project, and each Report can have multiple Pages in that Report.

Each Report has properties. To see the properties, select the Report and view the Property Panel.

The Page component is the base visual component upon which Drawing, Reporting, Standard, and Barcode components are placed. This is where all the designing and layout of a Report are completed.

The Page also has properties which can be viewed by clicking in a blank area somewhere on the page.

Global Page

The Global Pages are located under the Global Page Catalog node in the Project Tree. These Global Page Definitions are used as Master Page definitions. These Pages can also be Mirrored. Global Pages can contain Page layouts for things like letterheads, forms, watermark designs, and other Page layouts that can serve as a foundation for several Reports in the Project. An example would be mirroring a Global Page where you wanted to print the same contents (link an "Invoice"), but you wanted to put a different caption at the bottom of the Pages like "Original", "File Copy", and "Shipping".

Global Pages also have properties. Select the Global Pages from the Project Tree and look at the Property Panel to see all the properties.

Data Objects

Data Objects are the Data Connection components used to connect to data, or components used to control who is allowed to see which Reports from the Reporting Server.

Clicking on the New Data Object button on the Project Toolbar creates each Data Connection. After clicking on the button, the Data Connection dialog will appear. From this point, one of five selections can be made. The Data Object component choices available are DataLookupSecurity Controller, Database Connection, Direct Data View, Simple Security Controller, and SQL Data View.

The details of each Data Connection component will be covered in the next few sections. Once a Data Object is selected, the Data Object component will be placed in the Data View Dictionary in the Project Tree. Selecting the component and looking in the Property Panel will reveal it's associated properties.

Database Connection

The Database Connection component is the Data Object used to connect to data. This component can be added to a Project by clicking on the New Data Object button on the Project Toolbar, then selecting Database Connection from the Data Connections dialog.

Once the Database Connection is chosen from the Data Connection dialog, the Database Connection Type dialog will appear. This is where the type of connection that will be used to connect to that data will be chosen. In this image there is only one selection, but depending on what types of connections you have, there could be many more selections. After choosing the right data connection type, click Finish.

The Database Connection component, like all other components, does have properties. Select the component from the Project Tree and see the properties in the Property Panel.

Security Components

Using a Security component can control security to individual Reports. TRaveBaseSecurity cannot be used itself, as it is an abstract. However, you can create a descendant or TRaveBaseSecurity to implement your own security scheme or use one of the pre-built TRaveBaseSecurity descendants included with Rave. Rave currently includes two TRaveBaseSecurity descendants: SimpleSecurity and LookupSecurity.

To use a Security component, create one and then set the Report's SecurityControl property to the Security component instance that you want to control access to the Report.

Security components currently only affect Reports when served via the Reporting Server. When an unauthenticated user attempts to access a Secured Report, HTTP authentication will be used to authenticate the user.

Simple Security Controller

TRaveSimpleSecurity implements the most basic form of security by using a simple list of username and password pairs in the UserList property. UserList contains one username and password pair per line in the format:

Username=password

CaseMatters is a Boolean property that controls whether or not the password is case sensitive. Username is always case insensitive.

Data Lookup Security Controller

TRaveLookupSecurity allows username and password pairs to be checked against entries in a database table.

DataView specifies the DataView to use to lookup the username and password.

UserField is the field that is used to look up the username. PasswordField is the field that contains the password to verify against.

SQL Data View

SQL Data Views are used for creating self-contained DataViews to SQL databases. A database is specified using the Database property.

Parameters can be specified using the Params property.

The SQL to use is entered into the SQL property. At design time, a property editor for the SQL property invokes a visual query builder.

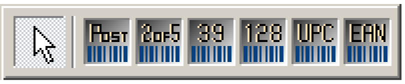
Chapter 16

Bar Code Components

In this Section:

- Describes how to place Bar Codes
- Describes how to encode Bar Codes.
- Gives brief descriptions of the Bar Code types.

Bar Code Component Basics



Bar Code components are used to create many different kinds of Bar Codes in a Report. Bar Codes are for users who know exactly what they need, as it requires background knowledge about Bar Codes and how they are used. It is not expected that the beginning user would have the background to use these components.

To place a Bar Code, click on the needed Bar Code component button and click on the Page.

To define the Bar Code value, go to the Property Panel and type the value into the Text property box. For Bar Codes containing check characters, please do not enter the check character, as it will be calculated automatically.

Brief Bar Code Descriptions

PostNetBarCode

PostNetBarCode uses the POSTNET (POSTal Numeric Encoding Technique) bar code and is specifically used by the Postal Service. It encodes ZIP Code information on letter mail for rapid and reliable sorting by bar code sorters. The PostNetBarCode can represent a five-digit ZIP code (32 bars), a nine-digit ZIP+4 code (52 bars), or an eleven-digit delivery print code (62 bars). Be aware that for the Post Office to recognize the bar code as being valid strict adherence to the guidelines must be met. Current standards require that the five digit zip plus four be used plus the two digit carrier route, which is most often obtained from the first two digits of the street address. It is recommended that the user check with the Post Office to obtain a copy of their current guidelines.

Example PostNetBarCode: 85210304119

2of5Bar Code

2of5Bar Code (interleaved 2 of 5) is a numbers-only bar code; in the Property Panel it is labeled 2of5BarCode. The symbol can be as long as necessary to store the encoded data. The code is a high-density code that can hold up to 18 digits per inch when printed using a 7.5 mil X dimension. "Interleaved" comes from the fact that the digit is encoded in the bars and the next digit is encoded in the spaces. There are five bars, two of which are wide and five spaces, two of which are wide.

Example 2of5BarCode: 2632534

Code39BarCode

Code39BarCode is an alphanumeric bar code that can encode decimal numbers, the uppercase alphabet, and the following special symbols " _ ", " . ", " * ", " $ ", " / ", " ", and " + ". The characters are constructed using nine elements, five bars and four spaces. Of these nine elements, two of the bars and one of the spaces are wider than the rest. Wide elements represent binary ones (1), and narrow elements represent binary zeros (0).

Example Code39: CODE 39

Code128Bar Code

Code128Bar Code is a very high-density alphanumeric Bar Code. The symbol will be as long as necessary to store the encoded data. It is designed to encode all 128 ASCII characters and will use the least amount of space for data of 6 characters or more of any 1-D symbology. Each data character is made up of 11 black or white modules. The stop character is made up of 13 modules. Three bars and three spaces are formed out of these 11 modules. Bar and spaces vary between 1 and 4 modules.

Example Code 128: Code 128

UPCBarCode

UPCBarCode (Universal Product Code) has a fixed length of 12-digits and can only encode numbers. UPC was designed for coding products. The format allows the symbol to be scanned with any package orientation. The check digit is calculated so there is no need to enter it when typing the value into the Text property.

Example UPCBarCode: 712345678935

EANBarCode

EANBarCode (European Article Numbering System) is identical to the UPC, except for the number of digits. EAN has a length of 13 digits - 10 numeric characters, and 2 "flag" characters that are usually country codes, and a check digit. This Bar Code is typically used for Non-U.S coding. The check digit is calculated for you so you do not need to enter it when typing the value into the Text property.

Example EANBarCode: 3847348484584

Chapter 17
Advanced Components

In this Section:

- Use of the FontMaster and PageNumInit are explained.
- Details of some advanced components are given.
- Calc Component details are given.

FontMaster

Each Text component in a Report has a Font property. By setting this property, a specific font can be assigned to the component. In many cases it may be somewhat useful and necessary to set the same font properties of more than one object. Although this could be accomplished by selecting more than one component at a time, this method has a drawback. One has to keep track of which fonts have to be of the same typeface, size and style, which is not an easy task when there is more than one report. This is where the FontMaster comes into play. Apart from allowing the user to define a global font for various components, the FontMaster also allows the user to define standard fonts for different parts of the Report, like for instance the headers, body, and footers.

The FontMaster is a non-visual component (designated by the green color of the button). To use one, simply click on the button and then click anywhere on the Page Designer. Note that being a non-visual component, there will be no visual reference of it on the actual Page, and like other non-visual components, it can be accessed using the Project Tree.

As mentioned previously, the main purpose of the FontMaster is to set fonts. It has very few properties. As most other components, it has the DevLocked, Locked, Name and Tag properties. They also contain the most important one of all, which is the Font property.

To set the FontMaster property, use the Font property on the Property panel, click on the ellipse button to get to the Font Dialog box and select the font and size settings. Click OK when done.

Once the FontMaster component is set, linking it to a body of text is simple. On the Report select a Text/Memo component, then use the down arrow button on the FontMirror property in the Property Panel to choose a FontMaster link. Any component that has the FontMirror property set to the FontMaster will be affected by and change to the FontMaster's font property. This allows the user to change fonts on various Text components at once and at the same time keep things consistent across the Report (when required).

It is important to note that by setting the FontMirror property on a component, the Font property will be overridden by the Font property of the FontMaster. This means that if a text object has a Font setting of Font A, by assigning a FontMaster to it with Font B, the Text component will automatically be assigned Font B and it's Font property will be ignored. Another side effect of using the FontMaster is that the Font Toolbar is disabled when the FontMirror property is set for that component.

There can be more than one FontMaster per Page; however, it is good practice to usefully rename a FontMaster. The Project Tree image shown previously has three FontMaster components on the Page. Two of them begin with FM, for FontMaster, and they follow with the name and size of the font that they represent. This is one naming convention that can be used to make the name descriptive and useful to the user.

PageNumInit

PageNumInit is a non-visual component that allows the restart of page numbering within a Report. Using a PageNumInit is similar to other non-visual components. It is used when more advanced formatting is required.

An example of using PageNumInit is in a CustomerStatement Report of a checking account. Account statements that customers receive every month may vary in the number of pages. In the monthly statements, the first page can define the account summary page layout, the second can define the customer's credits/deposits, and the third the withdrawals and debits. The first two pages may only be one page, but if the account activity is high for the customer then a section like withdrawals could be several pages. If the user producing the report wants each section numbered individually, the summary and deposits would have the pages marked "1 of 1" for both. For Withdrawals, an active customer account could have three pages of withdrawals and debits. Since this is a different section in the statement, it needs to be marked "1 of 3", "2 of 3", and "3 of 3". Using this kind of multiple page numbering is where PageNumInit comes in handy.

There are a couple of steps that must be completed to use the component efficiently. First define the Report Pages as usual. Add a DataText component from the Report Toolbar to the Page, placing it where the Page requires a Page number to appear.

Once it is in place correctly, in the Property Panel of the DataText component, click on the ellipse button in the DataField property.

This will open up the DataText Editor. Click on the Report Variables down arrow button and select Relative Page and click on Insert Report Var button to use the variable on the Report.

In the DataText area, "Report.RelativePage" should appear. After the text type in the following: + ' of ' +

From the ReportVariables drop-down list select TotalPages (choose it and press the Insert Report Var button). This will add the remaining text "Report.TotalPages" to the Data Text edit box area. When finished click OK.

The Property Panel now displays the sentence that was entered using the Data Text Editor.

Expanding the DataText component, the text from the editor will also appear.

To initialize the Page to a desired Page number, select PageNumInit from the Project Tree. On the Property Panel, type the desired Page number into the InitValue property box.

For every Report definition created in a Project, the PageNumInit will allow each report definition to be numbered independently of another.

DataCycle

The DataCycle component is an invisible data-aware band that would be used to control iterating information from a DataView. In general, a DataCycle component would be used to "Loop" or "Cycle" through the detail records until a change occurs at the master record level.

For example, imagine a bill being generated for multiple customers and each customer has many different purchases associated with them. For this bill you need to have each person's purchases on their own billing page, which has all and only their own listing of purchases. The table design for this might look like the following tables, where there is a master table listing of all the unique customers, and each customer has their own purchasing table, with all their own purchases listed.

The bill being created will need to have each unique customer ID number printed on one page as well as the items purchased on that page. Thus, the report will have to cycle through the table containing all the purchases made before proceeding to printing the next customer and the associated purchased items.

The main table containing the list of customer ID's would be found in what is called the MasterDataView. In the Property Panel, this master table name would be chosen from the property called MasterDataView. Once that is chosen, we will need to tell Rave what key will link the two tables together. This key is called the DetailKey. The DetailKey, in our example, would be the Customer ID's. Next, the table that will be cycled through, which is the table containing purchases to be printed, will be the DataView table. So, in essence the MasterDataView (driving table) and the DataView (details table) are connected by the Detail Key (the common unique key in each table).

The DataCycle can also be limited, or sorted while the Report is being generated. For example, supposed you just wanted to create reports for the Customers in Arizona. To limit Report creation to just the Arizona customers, the MasterKey will have to be set.

Setting the MasterKey is done by clicking on the ellipse (…) button next to the Property in the Property Panel. The Data Text Editor will appear. In the Data View area, choose the "Selected" radio button and then select the appropriate MasterDataView table, which is the Master Table in our example, by using the drop-down menu. Once you have done that, next select the field that will be used as the filter of the Master Table, or MasterDataView table. Once selected, click on Insert Field that is just underneath the Data Field drop down menu. This will place the appropriate text in the Data Text area at the bottom of the Editor. Now in the Data Text box, type in "='AZ'" to limit that field to just process the AZ customers. Click OK when done, and that will filter out non-Arizona customers.

DataMirror Section

The DataMirrorSection component is a section that will "Mirror" other "Sections" based on the contents of a "DataField". Mirroring Sections allow this component to be very flexible. Remember that Sections can contain any other component including graphics, regions, text, etc.

An example using this "DataMirrorSection" component is included in the RaveDemo project file. This example shows how to have a single report produce different envelope formats when sending to International or US addresses. The template for the International customers includes the country line and is be centered on the envelope, while the US format does not have the country line and is offset to the right of center and lower on the envelope.

This example shows several techniques on how to use Sections and Mirrors. On page 2 there are three Sections. Section 3 has the common address lines used by both templates. Section 1 and 2 contain Mirrors to Section 3. The International Section has the additional country line added in its template. Note, that "remarks" have been added to the page 2 design and they are not part of any of the sections. The "US Template" and "International Template" are just plain Text remarks that assist in the purpose of each Section shown.

Once understanding how the Master Templates were done, go to Page 1, and select the "DataMirrorSection". Notice the "DataView" and "DataField" properties have been set to the field that will be used to select which Format Template to Mirror. To examine that logic, go to the "DataMirrors" property and click on the ellipsis to open the Data Mirror Editor. Select each "DataMirror" item and note the settings in the bottom portion of the dialog box. This example has two "templates", but feel free to add more to fit more complex reporting needs.

NOTE:

Normally one of the settings will be defined as the default. If a default is NOT defined and the field value does not match any of the other settings, then the format used will be the normal contents of the DataMirror Section component.

CalcOp

The CalcOp component is a non-visual component that allows an operation (defined by the Operator property) to be performed on two values from different sources

(Src#CalcVar, Src#DataField or Src#Value). The result can then be saved in a project parameter like CalcTotal (through the DestParam and DisplayFormat properties).

For example, there could be two DataText components that need to be calculated together. Like *A+B=C*, where A and B represent the two DataText component values and C represents the result. This is where the CalcOp component would be used.

To choose the DataText components in the above example, use the Src1X and Src2X properties, where X is either CalcVar (a calculation variable), DataField, or DataValue. These are the three types of value sources that can be used for calculation in a CalcOp component. Note that the Src (source) can only use ONE of the three available source types, and there can only be two sources, Src1 or Src2.

The three source types have many different values associated with them. For a CalcVar source, the drop down menu will list all the calculation variables available in that Page available for use. This calculation variable is just that, a variable for holding a value. This value can be from another CalcOp component or from some other calculation component. For the DataField source to get available values first designate what DataView the DataField will come from. In other words the DataField is a field in a table, which is the DataView. So, in order to choose a field, the DataView must first be chosen. For a Value source, typing in a numeric value is all that is needed to fill this property. Again, remember only one of these three sources can be assigned to a source.

Once the sources are chosen the operation to be used between them has to be chosen. To choose the desired operation, use the Operator property by using the drop down menu and making the appropriate choice. In the example, *A+B=C*, the operator would be "coAdd".

There may be times when a function needs to be performed on a value before it is processed with the second value. This is where the property SrcYFunction, where Y is 1 or 2 for Src1 or Src2, will become handy. With SrcYFunction, the value can be converted (like from hours to minutes), or have a trigonometric function performed on it (like take the sin of the value), or have other functions performed on the value (like take the square root of the value, or take the absolute value).

Once the two values are chosen, it is now time to deal with the result. In the example, *A+B=C*, C is the value of the result. The result can either be written out to a project parameter (most likely a DataText component) or just held as an intermediate value. To write out the value to a parameter, use the DestParam property to choose the desired parameter to write the result write to. If the result is going to be in intermediate result, there is nothing else to do to the component. Although, it is highly suggested to rename the component, by using the Name property, to easily recognize the component for future use. This is because, as in intermediate values, you will most likely use the components in another CalcOp component, and a good reference will help to easily create the next step in the calculation process.

After setting the values and setting the result destination, it may be necessary to format the result or to even perform one last function on the result. Using the DisplayType and DisplayFormat properties to format the result into any necessary format. The DisplayType has two options, DateTimeFormat and NumericFormat. DisplayFormat has many options that must be typed into the edit box. To find these options look at the Formatting Appendix in the back of this manual. Like the two values, A and B, in the example, *A+B=C*, the result C can also have a function performed on it before being written out to a parameter, or as a holding value. To select a function, use the ResultFunction property drop down menu.

The CalcOp components can also be chained together for more complex expressions using the Src#CalcVar properties, which can be set to other CalcOp or CalcTotal components. For example, to create the complex expression shown to the right, it will be necessary to break up the expression into simple 2 value steps.

To break up the complex expression, four new expressions will need to be created and saved as CalcOp components. So the resulting components, Z, Y, X, and W will be four separate intermediate CalcOp's. But, as you can see from the flow of the expression, Z and Y will have to be completed before X can be created, as it is dependent upon the two previous operations. And finally, W will be completed last, after X, to get the final correct value.

$$\frac{(A+B)*(C*D)}{0.80} = E$$

$$\begin{aligned} &A+B=Z \\ &C*D=Y \\ &Z*Y=X \\ &\frac{X}{0.80}=W \qquad = E \end{aligned}$$

Just as it is important to do the calculations in order, it is important to make sure the components are in order in the Project Tree. When a report is executed, the report executes components down the Project Tree. For CalcOp components or any calculation component, this means that they need to be in the correct order. In the example above, if Z, Y, X, and W were in the Project Tree, Z would have to be first in the tree and W would have to be last in the tree. This would mean that at execution, Z would be processed first, then Y, and so on. It is also important to note that if Z is dependent upon any other components (like other CalcOp components or DataText components), those components must come first in the Tree before the Z component is processed. Making sure the components are in correct order is known as setting the print order. To move the components up and down within the Project Tree, use the Alignment Palette to change the print order of the components.

CalcController

CalcController is a non-visual component that acts as a controller, along with DataBands, for CalcText and CalcTotal components through their Controller properties. When the controller component is printed, it signals all calculation components that it controls to perform their summation operation. This allows totals to be performed on group bands, detail bands or whole pages depending upon the location of the CalcController component. Another feature of the CalcController component is its ability to initialize a CalcText or CalcTotal component to a specific value (through the InitCalcVar, InitDataField and InitValue properties). A CalcController component will only initialize values if it is used in the Initializer property of CalcText or CalcTotal property.

CalcTotal

The "CalcTotal" is a non-visual version of the CalcText component. When this component is "printed", its value is typically stored in a project parameter (defined by the DestParam property) and formatted according to the DisplayFormat property. This can be useful when performing totaling calculations that will be used with other calculations before being printed. Leave DestParam blank if the value of CalcTotal will only be used by other calculation components such as CalcOp.

Exercise: Using Font Master

1. Create a new page in the report. Select New Report Page from the Project window.
2. Go to the Project Tree and select the Page.
3. In the Property Panel, type in "FontMaster" in the Name property, while the Page is still selected.
4. Go to the Standard Toolbar. Find the FontMaster component and click on the button. Then click on the Page.
5. Look at the Project Tree Panel and notice that there is a new component underneath the Page called FontMaster1. But, when looking at the Page, there is no visual component. FontMaster is a non-visual component.

6. Make sure the FontMaster component is selected by clicking on it in the Project Tree. The component will be highlighted in the Project Tree to indicate that it is selected.
7. While the FontMaster component is selected, look at the Property Panel. In the Font Property, click on the ellipse button (the button with three dots).
8. The Font Dialog will appear after clicking the ellipse button. Use the scroll buttons and check mark boxes to make your desired selections. For this first FontMaster component, select Arial Font, Bold Font style, 14 size, and Underline in Effects.
9. Look at the Property Panel, with the component still selected. The Font Property will reflect the selection made in the Font Dialog.
10. Now with the Font Property still selected, scroll to the Name Property and put "FMArial14BldUndrln" in the Name property edit box.
11. Next look at the Project Tree and notice the name change. It becomes very useful and helpful to rename the components in the Project Tree in order to distinguish each component from each other. This is why we demonstrate renaming of the FontMaster component and the Page component.
12. Now, complete steps 4 to 11 three times using the following names and settings. FMArial16ItlcUndrln: Arial Font, size 16, Italic, an Underlined. FMTimesNwRmn12: Times New Roman Font, and size 12. FMCourier12: Courier Font, and size 12. To make understanding the renaming of many components, sometimes it is helpful to use a specific naming convention. In these examples we use the following naming convention for the FontMaster component names:
13. Next, drop down five Text components on the page, as well as one Memo component.
14. We will pretend to write a "letter" using components dropped on the Page.
15. Use one Text component to use for a date holder, and use the other four Text components to make an address label. The Memo component can be used for the body of the message. So, fill your components appropriately. For more information on how to fill the components with your own text, see the Property Descriptions Listing in the appendix, or see the chapter on Standard Components. Feel free to use the alignment tools to get the "letter" components to align correctly. Also, after some of the following steps it may be necessary to resize the text components to display all the text correctly.
16. Next select the Text component that has the date. While selected, change the FontMirror Property to FMCourier12 by using the drop down menu. Notice that the font of the Text component changed to the pre-set font settings of the FMCourier12.
17. Next select the four address Text components. The Property Panel will show the designation that this is a multiple selection of Text components. In the FontMirror property, choose FMTimesNwRmn12.
18. Select the Memo component, which contains the body of the "letter". While it is selected, choose FMArial16ItlcUndrln in the FontMirror drop down menu.
19. This ends mirroring of fonts to components on the Page. One last thing to cover is what to do when you change your mind about the fonts?
20. Select all the address Text components. Change the FontMirror property to FMArial14BldUndrln. Now, this would be a way to change many Text/Memo components from one font setting to another, without changing each Font property in each component.
21. Sometimes there may be many components linked to one FontMirror component, because they all relate to one specific area of the report. In this example, we can assume that all address headers would be linked to the FMArial14Bldunderln Font Mirror property. So, to change this we would have to redo the one FontMirror property.
22. Select the FMArial14BldUnderln FontMirror component in the Project Tree.
23. In the Property Panel, while the component is still selected, click on the Font property ellipse button.
24. Change the Font Dialog properties to the following: Times New Roman, size 10 font, with no other Font Effects.

25. In the Name property of the FontMirror property, put the following name: FMTimesNwRmn10.

Exercise: Setting up PageNumInit for Page Numbering

1. First create a new Page, or use what is left from the previous example, as we will do here. If this is a new Page, drop some Text components on the Page. These are just filler components in this example.
2. Go to the Report Toolbar, and find the Data Text component. Click on the Data Text component and then click on the Page.
3. Place the DataText component in an area where a Page number might appear. For example, place the DataText component at the bottom of the Page.
4. Make sure the DataText component is placed correctly, and make sure it is still selected. Check for selection by the visual appearance of the green pip surrounding the component, or by the component being highlighted in the Project Tree.
5. While the DataText component is selected, go to the Property Panel and look for the DataField property. Click on the ellipse button.
6. The Data Text Editor will next appear. Look for the *"Report Variables"* section in the Editor.
7. There will be an arrow at the end of the Report Variables selection window. Click on it to see all the selections available. Use the scroll bar to move up and down through the menu.
8. In the Report Variables section, look for *"Relative Page"*. Click on it to select.
9. Next click on the Insert Report Var button to the right of the Report Variables section. This will place the appropriate text into the Data Text area at the bottom of the Data Text Editor.
10. Go to the Data Text area and type in the following: + 'of' + . This will include spaces before and after each symbol.
11. Now return to the Report Variables drop-down list. Scroll to look for TotalPages. Select it when you find it.
12. Click the Insert Report Var button, to get the remaining text into the Data Text area. "Report.TotalPages" will be added to the Data Text area.
13. Click OK when done. Your DataText area will look like the following.
14. After clicking OK, you will be returned to the Designer and to the Property Panel. If you expand out the Property Panel and look at the DataField property, you will see the results of the DataText are from the Data Text Editor.
15. On the Page, the DataText component will probably look like the DataText component labeled A below. If you expand the borders of the DataText component (as seen in B), you can see that the results of the Data Text Editor appear. There is really no need to expand the component; it is just for you to see in this exercise.
16. There is one more thing left to do to complete the page numbering, we need to initialize the Page to the desired page number. So, go to the Project Tree and find the PageNumInit component. Select the component when you find it.
17. While PageNumInit component is still selected, go to the Property Panel and find the InitValue property.
18. In the InitValue property, type in 1. This will mean that our pages will begin with page 1.
19. That is it. That is all that is needed to setup the PageNumInit. Note that in one project, with this feature, each Report can be number independently of each other.

Chapter 18
Adaptable Reports

In this Section:
• Explains the problems related to adaptability. • Explains how to use Anchors to "fix" your components. • Shows how to work with waste areas on different printers.

Overview

One of the biggest problems that designers face when creating reports is adaptability. What is adaptability exactly? First, when a report is created, the process normally takes place on a specific computer, which has one or more printers connected to it. When that report is executed in the field, however, factors such as paper size, orientation and waste areas all come into play. Secondly, reports should allow for a certain amount of flexibility in the overall design and appearance of the layout of the report.

Rave helps overcome these problems by giving the designer access to particular properties of the report. In this chapter, these properties will be explained in detail and will allow the resulting reports to have a standard look over various platforms. This way, the designer can focus more on creating reports than worrying about deployment difficulties. The reports will also be able to adjust layouts based on external parameters passed in during execution.

Anchors

Most visual components have a common property called Anchors. This powerful feature defines how the object will move when its Parent is resized. There are two Anchor values that are set, one for the horizontal and one for the vertical. The default Anchor is set to the top left corner of the Parent control. What this means is that when the Parent (e.g. a Section component) is resized, the component will stay the same distance from the left and top corner of the Parent. Two other Anchor values, Bottom and Right, are very similar in function (they will Anchor to the bottom right corner of the Parent).

The Anchor setting of Center will Anchor the Child component to the Parent's center. Stretch will actually resize the Child component so that it's sides stay the same distance from the Parent. Stretch is most often used when you want a Child component to always match the width or height of the Parent. Two other special Anchor settings, Resize and Spread are useful for groups of components. Resize will proportionally resize the components and the spaces between them as the Parent is changed. Spread will proportionally resize the spaces only (the components will stay the same size) as the Parent changes. Drop down a Section component and place a few rectangles or Text components inside and change the values of those components to see how the Anchor property can affect things.

Anchors can be used to create adaptable Reports when combined with other Reporting features. Imagine that a Report needs to be defined that can be printed in either landscape or portrait orientation or that the Report may be run on different size papers. Setting the Anchors properly will allow one Report to adjust to these changing conditions. A typical table listing Report will be composed of a Region component; it's Band components and the Text and DataText components inside each band. If the Region component is set to Anchor Stretch on both vertical and horizontal and the Text and DataText components are all set to a horizontal Anchor of Resize, the Report will adjust to any of these changes.

Other changes, such as the printer's individual waste area, can also be solved with Anchors and will be discussed in the next section.

Waste Fit

Most printers have a Region on the paper that is called the waste area. What this represents is the portion of the Page where no printing can be done because of restrictions of the actual printing hardware (usually because the printer uses that region of the page to "grab" a hold of the paper when it is feeding it through the rollers). These values may range from 0 (common for dot-matrix printers) to 1.5 inches. Ink jet printers typically have large bottom margins (and sometimes a 0 top margin), while most ink jet and laser printers have about a 0.25 inch left and right waste area.

The red-dotted lines that run on the border of the Page represent the waste-area of the currently selected printer (or specific values if the preferences are set that way). Anything in between the red line and the border of the Page is the actual waste-area of the printer and you should avoid placing components within that area. The problem is how to know what the actual waste area of the destination printer will be. If the Report is designed with extremely large margins that will be inside the waste area of all printers, then a lot of unnecessary blank space will be on each Page. The WasteFit property is Rave's answer to this common reporting problem.

The Page has a property called WasteFit, which can have one of two values, either true or false.

By default the value is False. Setting the property to true will make the Page adjust the components it contains so that they fit within the waste-area of the destination printer. The components contained will be resized so that they all fit in and adjust accordingly.

However, it is not sufficient with setting the WasteFit property to True for all this to happen automatically. Child components should have their anchor property set accordingly so that they too can adjust accordingly to the changes of the Page.

When used properly, this property is a very powerful feature that allows the reports developed to automatically adapt to different destination printers.

Editor Anchor

Normally, ReportBands/columns/paragraphs are justified to the Left and Top of a design area. This will be fine for the majority of Reporting needs.

However, have you ever wanted to control the starting or ending position of your report components dynamically? This could be a two-column Section, where the left column is a Memo component which changes in height and the right column is a Text component, which is always a one line item. How do you make both components "float" and align to the bottom of the design block? In Rave this is done with the Anchor property.

To change the Anchor style go to the Anchor property of the desired component and click on the ellipse symbol, it will open the Anchor editor like the one shown above. This provides a method to select the Anchor style wanted for that component by using the appropriate vertical and horizontal radio button. Note that a representation appears below each Anchor selection to give a visual indication of what that setting is designed to accomplish. The last three settings, Stretch, Resize, Spread are a little difficult to explain, but view the sample and there will be a picture showing the differences between the settings.

An important point to note is that the Anchor property settings are relative to that component's Parent. So, if the Parent is a Page, then the settings are relative to the Page sides. If the Parent is a Section, then the settings are relative to the section borders. One way to visually see the parentage is to examine the component(s) on the Report Node in question in the Project Tree. Go up one level and that is the Parent.

The following combinations are not restrictive, but normally the Anchor settings will be paired as follows:

- Left/Top justified
- Right/Bottom justified
- both Center justified
- both Stretch
- both Resize
- both Spread

Chapter 19

Batch and Chain Reporting

In this Section:

- Discusses how to Batch Reports
- Covers calling and Page Chaining

Batch Pages

Probably the most common linking of Pages would be a batch processing sequence that defines a list of independent Page definitions. The first Page in the list runs to completion, then it calls the second, which runs to completion, and so on until the last Page of the defined sequence has been completed. The main thing to remember with batch Pages is that each Page definition is independent of the others and runs to completion before the other Page starts. Remember that the requirement to do batch processing is to simplify the administrative tasks when running a group of Page definitions on a recurring basis.

Define the sequence of Pages to print through the PageList property, which is available at the Report node level. When this property is selected (click on the ellipsis button), it will open a dialog window that will build the list from the Page definitions within the selected Report node. The advantage of this method is that the PageList is independent of the individual Page definitions. This means that the PageList would run the whole sequence of Pages (Reports), but individual Reports can still be selected individually and run by itself without invoking the batch link. Remember, Report Pages are not visible to other Reports, so if flexibility of calling separate Page definitions is needed, define the reusable Pages as global Pages instead of Report Pages.

WARNING
Another method to do Batch Pages is to set the first Page definition GotoPage property to the second Page definition name. Now, when the first Page is completed, it will automatically start the second Page definition. The problem with this technique is that anytime the first Page is run it will ALWAYS start the second Page as they are linked together at the Page definition level.

Calling Pages

(Page Node - GotoMode property - gmCallEach setting)

Another type of linking would be Reports where there must be a particular flow of Pages. The easiest example of this would be a Report where each record ALWAYS produces three Pages of output. This could be a "form Report" where Page 1 has a patient's demographic information, Page 2 has medical information and Page 3 has insurance information. So, when going through the database, each patient's record would produce 3 Pages of output. See the Exercises for more step-by-step instructions on how Call Pages.

Chain Pages

(Page Node-GotoMode property-gmGotoDone setting)

A chain of Pages is similar to a Batch. The Batch technique discussed earlier used the PageList property and cautioned you about using the GotoPage property. However, if you plan ahead, then the use of the GotoPage property with the Pages from the Global Page Catalog is very powerful. The exercise at the end of this chapter does a Multi-Page definition that includes an invoice, file copy, shipping document and packing slip.

Different First Page Format

(Page Node-GotoMode property-gmGotoNotDone setting)

Another type of linking would be Reports where the first Page format is different than the remaining Pages. This could be a Report that has a title layout maybe even with a company logo on Page 1, but the remaining Pages are all the same design layout. What you need to do for this is to create the Page 1 definition (first Page) that points to Page 2. Then create the Page 2 definition (all remaining Pages) that points to blank. The Exercise at the end of the chapter goes through the detail steps to complete this process.

Different Odd/Even Page format

(Page Node-GotoMode property-gmGotoNotDone setting)

Another type of linking would be Reports with a format based on an odd/even Page definition. This could be Reports that are going to be printed on both sides of the paper (duplex style) and have holes punched in one side, so that final Report could be put in a binder. This would mean that the inside margin (say 1 inch) would be larger than the outside margin (½ inch). To create a loop where the Page 1 definition (odd Page) points to Page 2 and will call it if the Report is not done when it gets to the end of the Page AND the Page 2 definition (even Page) points to Page 1 and will call it if the Report is not done at the end of that Page. This loop will need to continue until Report is completely printed. See the Exercise at the end of this Chapter to get detailed steps to complete this task.

Batch / Chain Reports

Once the Report Design is complete and working, the day-to-day routines get settled. At this point, it would not be unusual to find out there often is a need to print several Reports in some kind of sequence. This could be a group of Reports that are always generated at the end of each month, quarter, etc. Another example might be a series of executive summaries that are needed on a demand basis by upper level management. Of course, it is possible to run each required Report individually for these repetitive cases. Then when they are all done, group them together and give them to the requesting office. This often means that there is a checklist that details what repetitive Reports are required, when and for whom.

This problem of producing a repetitive sequence of Reports is solved with Rave by linking the Reports to each other. Rave has the ability to link the Report Page definitions in a wide variety of ways by setting different combinations of three properties; Page.GotoMode, Page.GotoPage and Report.PageList. It is important to note that the setting of the GotoMode property determines the behavior of the GotoPage property. The PageList property is at Report Node level and is designed to provide a way to initiate several different GotoPage property chains in a defined sequence.

Although we will give examples of how we envision achieving various Batch/Chain sequences of Pages, it will not be possible to include all of the possibilities. The secret is that one should understand and be flexible on how to "mix and match" the various parts of Rave. In particular, pay special attention to the Global Page catalog and the GotoMode, GotoPage, Mirror and PageList properties. Different combinations of these provide a wide range of output options. There is a lot of power here, so the best advice is to start with simple Reports. Then add some of these powerful extras and as they make sense.

Exercise: Calling Pages

The way to accomplish this would be:

- Complete all normal definitions for the Page 1.

- Complete all normal definitions for the Page 2.
- Complete all normal definitions for the Page 3.
- Set the GotoPage property of the first Page to point to the second Page.
- Set the GotoMode property of the first Page to gmCallEach setting.
- Set the GotoPage property of the second Page to point to the third Page.
- Set the GotoMode property of the second Page to gmGotoDone setting.
- Insure that the GotoPage property of the last Page definition is blank.

The gmCallEach setting of the Page 1 GotoMode property will activate the GotoPage property after the first physical Page has printed. So, at the end of physical Page 1, Page 2 definition will be started. Because the Page 2 definition has a gmGotoDone setting, it will go to the Page 3 definition when it has finished the printing the second Page. The Page 3 definition will print that Page's definition and then return control to the Page 1 definition because it's GotoPage property is blank. You can join any number of Pages together in this manner. The key is that a gmCallEach setting will save off the calling Page so that whenever a blank GotoPage property in the Page chain is encountered, it will continue with that calling Page.

Exercise: Chain Pages

For this example, we are going to do a Multi-Page definition that includes an invoice, file copy, shipping document and packing slip.

- Create a section on a Global Page that will be mirrored later.
Complete all of your normal definitions for the Invoice Page within this section.
- Create a section on a Global Page that will be mirrored later.
Complete all of your normal definitions for the File Copy Page within this section.
- Create a section on a Global Page that will be mirrored later.
Complete all of your normal definitions for the Shipping Page within this section.
- Create a section on a Global Page that will be mirrored later.
Complete all of your normal definitions for the Packing Slip Page within this section.
- Create a "New Report" that will hold the definitions for these Pages.
- Create a new Page definition, "Invoice"
drop a section on this Page definition and set the Left and Top properties
mirror the section to the "Global" Invoice Page section
drop a text component on the bottom of the Page "INVOICE COPY"
- Create a new Page definition, "FileCopy1"
drop a section on this Page definition and set the Left and Top properties
mirror the section to the "Global" FileCopy Page section
drop a text component on the bottom of the Page "FILE COPY 1"
- Create a new Page definition, "FileCopy2"
drop a section on this Page definition and set the Left and Top properties
mirror the section to the "Global" FileCopy Page section
drop a text component on the bottom of the Page "FILE COPY 2"
- Repeat those steps, for the "Shipping Document" and "Packing Slip"

- Now go back to the "Invoice" Page definition (NOT the global definition"
Set the PageMode setting to gmGotoDone
Set the PageGoto to point to the "File Copy 1" Page definition.
- Repeat these steps for each of the copies, remember to leave the last one blank

The mirroring of a Global Page offers a lot of "reuse" of a Master Design. This example showed that the "FileCopy" could be mirrored twice and "label" each Page definition differently.

Exercise: Different First Page

The way to accomplish this would be:

- Complete all normal definitions for Page 1.
- Create a "New Report Page" for the second Page definition.
- Set the GotoPage property of the first Page to point to the second Page.
- Set the GotoMode property of the first Page to gmGotoNotDone setting.
- Complete all normal definitions for Page 2.
- Insure that the GotoPage property of the second Page is blank.

The gmGotoNotDone setting of the GotoMode property will activate the GotoPage property after the first physical Page has printed but only when the current Page definition has not finished (for example, EOF - End Of File). So, at the end of physical Page 1, Page 2 definition will be started. Because the Page 2 definition does NOT have anything in the GotoPage property, the Page 2 definition will remain in effect for all remaining physical Pages until the Report is done.

Exercise: Different Odd/Even Page

The way to accomplish this would be:

- Make a Section on the first Page definition that is set for the "Odd" Page margins.
For example, set the properties Left = 1.0, Width = 7.0, Top = 0.5 and Height = 10.0.
- Complete all of your normal definitions for Page 1 within this section.
- Create a "New Report Page" for the second Page definition
- Drop a Section on the Page 2 and set the top and left margins for the even Page settings.
For example, set the Left property = 0.5 and Top property = 0.5.
- Set the Mirror property of Page 2 Section to point to the Section on Page 1.
- Set the GotoPage property of the second Page to point to the first Page.
- Set the GotoPage property of the first Page to point to the second Page.
- Set the GotoMode property of BOTH Pages to gmGotoNotDone setting.

The gmGotoNotDone setting of the GotoMode property will activate the GotoPage property after each physical Page has printed but only when the current Page definition has not finished (for example, EOF - End Of File). So, at the end of physical Page 1, Page 2 definition will be called. At the end of physical Page 2, Page 1 definition will be called. This loop will continue until one of the Pages is completed. If control would need to be passed to another Page at this point, use the Report components PageList property to select the next Page.

Chapter 20

Preferences

In this Section:
• Shows how to get to the Preferences Dialog • Explains each tab in the preferences dialog

Getting to the Preferences Dialog

Before getting started in Rave lets take a short detour and learn about controlling the appearance of the RAVE designer. Under the main menu, select "Edit" then "Preferences". This will open a Preferences dialog where the look and feel of the RAVE system can be customized.

Defaults Tab

"Defaults" tab provides a place to set many of the RAVE Visual Designer Project and Page settings.

Settings in the Default tab section will be used when a new Project or new Page is created. After a Project has been started, the Project properties can be changed at the Project level and then they will be saved for that Project. The same applies to the properties of Pages. For example, a Project could have a default grid of 0.25 inches, but on a special Page the settings could set the Grid Spacing to 0.1 inches.

Designer Tab

The "Designer" section controls changes of the Grid system, Alignment options, units of Measurement and Zooming.

The "Zoom Increment" controls the amount of change that is made with the in/out zoom controls on the Zoom Toolbar.

The "Alignment Options" section, controls the visibility of the Order buttons and Tap buttons. It also controls the 'speed key' assigned to the Tap Tools. The speed key is really a predetermined amount of distance that the component is allowed to move with one keystroke. To use this feature, select the component and click on any of the arrow keys on the toolbar. Or, use the arrow keys on the keyboard while pressing the Ctrl-key.

In the "Grid Settings" area, grid color and associated changes can be applied to the Page of the Report. "Snap To Grid" will cause the mouse cursor to move and size components in increments of the Grid Spacing. The "Draw Grid On Top" option allows the Grid to be seen visually on top of components that have a solid background color. To hide the Grid Lines, make sure that "Show Grid" is unchecked. Several of these preferences are controllable from the Designer Toolbar. For more information about the Designer Toolbar and other toolbars see the Utility Toolbar Chapter.

The "Waste Area" controls the visibility of the amount of excess between the area border (the red dashed line) and the edge of the Page. "Show Waste Area" hides/shows the red dashed border around the Page. Then next two selections control the Waste Area width. The Waste Area can be set to the default printer settings, or it can be set manually by using the Left, Right, Top, and Bottom edit boxes.

In the "Background Gradient" section, the three colors that make up the area between the Page and the borders of the program can be changed. Clicking on the color wheels and selecting the desired colors change the gradient of the background. To make the background a solid color, simply select the same color for all three color options.

The visibility of the Order Alignment and Tap Tools are controlled by settings in the Alignment Options section of the Designer Tab. To get to this section click Preferences in the Edit menu and choose the Designer Tab.

Also in the Alignment Options, the Tap Tools have a 'speed key' assigned to them. This speed key is really a predetermined amount of distance that the component is allowed to move in one keystroke. The "Tap Distance" is initially set at 0.01 inches. If this distance is increased it will take less time to move the component in the desired direction. To use this feature, select the component and hold the Ctrl key while clicking on any of the arrow keys in the toolbar. Or, use the arrow keys on the keyboard while pressing the ctrl-key.

Environment Tab

"Environment" contains settings for user viewed information, toolbars, and page environment options.

In "User Level", the user's level can be increased or decreased. This setting controls the amount of information to be displayed to the user, mainly dealing with the various properties that will be covered later in this manual. Do not be distracted with all the advanced properties at first. To simplify the early Rave experience, start out at either the Beginner or Intermediate User Level setting. While the Beginner and Intermediate settings are adequate for designing a Rave report, these lower settings will be missing some controls. Thus, if there are controls that are covered and it appears that Rave does not have these controls, check the User Level preference setting.

The "Options" in Environment section, controls the Band Headers, environment settings, and rulers. The "Always Show Band Headers" will have the Band Headers shown at all times. "Save Environment Only Changes When Exiting" will save all the changes made to the environment when the program is shut down. "Display Rulers" will keep the rulers visible around the Page if selected.

"Run in administrator mode," allows the user to be prompted for a password if there is a password in the file. When this selection is not selected, and there is a password in the file, the user will not be prompted for a password, the program will just run in non-administrator mode. An administrator can place a password in different areas to restrict access. Placing a password in the AdminPassword property restricts user access.

"Language" changes the language of the user interface. To have this option, the language file translators needs to be installed. Once installed simply choose a language.

Shortcuts Tab

The "Shortcuts" section allows the user to define keyboard shortcuts for a wide variety of Project actions.

It may save you some time to check the list of *"Actions"* to see if a shortcut already exists for those tasks that are repeatedly done while you are using the Rave designer. If there is no shortcut, then you may be able to create a shortcut that will be easy for you to remember.

Appendix A
Formatting

Below is a list of different format codes and what they will accomplish for each output type.

AlphaNumeric Items

Description: DisplayFormat formats the value given using the format string. The following format specifiers are supported in the format string:

	Format String	123456.78	-123.0	0.5	0.0
Examples:	#,##0.00	123,456.78	-123.00	0.50	0.00
	#.#	123456.8	-123	.5	0
	$,0.00	$123,456.78	$-123.00	$0.50	$0.00
	0.00;(0.00);'-'	123456.78	(123.00)	0.50	-----

Specifier	Represents
0	Digit place holder. If value being formatted has a digit where the '0' appears, then the digit is copied to the output string. Otherwise, a '0' is in the output string.
#	Digit place holder. If value being formatted has a digit where the '#' appears, then the digit is copied to the output string. Otherwise, nothing appears in that position.
.	Decimal point. The first '.' character in the format string determines the location of the decimal separator in the formatted value. The actual character used as a the decimal separator in the output string is determined by the Number Format of the International section in the Windows Control Panel.
,	Thousand separator. If the format string contains a ',' characters, the output will have thousand separators inserted between each group of three digits to the left of the decimal point. The actual character used as a the thousand separator in the output is determined by the Number Format of the International section in the Windows Control Panel.
E+	Scientific notation. If any of the strings 'E+', 'E-', 'e+', or 'e-' are contained in the format string, the number is formatted using scientific notation. A group of up to four '0' characters can immediately follow the 'E+', 'E-', 'e+', or 'e-' to determine the minimum number of digits in the exponent. The 'E+' and 'e+' formats cause a plus sign to be output for positive exponents and a minus sign to be output for negative exponents. The 'E-' and 'e-' formats output a sign character only for negative exponents.
'xx'/"xx"	Characters enclosed in single or double quotes are output as-is, and do not affect formatting.
;	Separates sections for positive, negative, and zero numbers in the format string.

The locations of the leftmost '0' before the decimal point in the format string and the rightmost '0' after the decimal point in the format string determine the range of digits that are always present in the output string.

The number being formatted is always rounded to as many decimal places as there are digit placeholders ('0' or '#') to the right of the decimal point. If the format string contains no decimal point, the value being formatted is rounded to the nearest whole number.

If the number being formatted has more digits to the left of the decimal separator than there are digit placeholders to the left of the '.' character in the format string, the extra digits are output before the first digit placeholder.

To allow different formats for positive, negative, and zero values, the format string can contain between one and three sections separated by semicolons.

One section: The format string applies to all values.

Two sections: The first section applies to positive values and zeros, and the second section applies to negative values.

Three sections: The first section applies to positive values, the second applies to negative values, and the third applies to zeros.

If the section for negative values or the section for zero values is empty, that is if there is nothing between the semicolons that delimit the section, the section for positive values is used instead.

If the section for positive values is empty, or if the entire format string is empty, the value is formatted using general floating-point formatting with 15 significant digits.

Date / Time items

Items that are either a date or time field can use the following format codes. The format specifiers are not case sensitive. If the format parameter is blank then the value is formatted as if a 'c' specifier had been given. The following format specifiers are supported:

Examples: dddd, mmmm d, yyyy => Monday, September 21 1998
d mmm yy => 21 Sep 98

Specifier	Displays
c	Displays date using format given by ShortDateFormat global variable, followed by time using format given by LongTimeFormat global variable. The time is not displayed if fractional part of the DateTime value is zero.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat) using the strings given by the ShortDayNames global variable.
dddd	Displays the day as a full name (Sunday-Saturday) using the strings given by the LongDayNames global variable.
ddddd	Displays the date using the format given by the ShortDateFormat global variable.
dddddd	Displays the date using the format given by the LongDateFormat global variable.
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec) using the strings given by the ShortMonthNames global variable.
mmmm	Displays the month as a full name (January-December) using the strings given by the LongMonthNames global variable.
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).

s Displays the second without a leading zero (0-59).

ss Displays the second with a leading zero (00-59).

t Displays the time using the format given by the ShortTimeFormat global variable.

tt Displays the time using the format given by the LongTimeFormat global variable.

am/pm Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.

a/p Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.

ampm Uses the 12-hour clock for the preceding h or hh specifier, and displays the contents of the TimeAMString global variable for any hour before noon, and the contents of the TimePMString global variable for any hour after noon.

"/" Displays the date separator character given by the DateSeparator global variable.

: Displays the time separator character given by the TimeSeparator global variable.

'xx'/"xx" Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

Appendix B
Keyboard / Mouse Shortcuts

Below is a list of different Keyboard / Mouse combinations that can be used as a shortcut. See Preferences - Shortcuts for assigning keyboard keys to your own shortcuts.

Page Designer or Project Tree

Click	on a component selects that component
Right Click	shows context menu for that component
Shift Alt Click	adds all components of the same type as the component clicked on to the selection list of the current page designer
Shift Ctrl Click	adds all children of clicked component to selection list
Shift Click	on a component toggles the selection for that component. This can be used to select multiple components.

Page Designer Only

Click	in blank area of Page Designer removes selection of all components
Ctrl + Arrow Keys	taps (moves) selected components in direction of arrow key
Ctrl C / Ctrl Ins	copies selection to clipboard
Ctrl Click	centers the design window to location clicked
Ctrl F4	unloads current global page
Ctrl V / Shift Ins	paste clipboard to page designer
Ctrl X / Shift Del	cuts selection to clipboards
Delete	deletes currently selected component(s)
Escape	changes selection to parent of current component
F9	executes the current report
F11	toggles between page designer and property panel
Shift + Arrow Keys	changes size of selected components (Up = decrement height, Down = increment height, Left = decrement width, Right = increment width)

Project Tree Only

Alt Drag	DataField component to page designer - creates text component
Alt Drag	selected component to container component in Project Tree - makes the destination component (must be a container component like sections or regions) the parent of all selected components
Ctrl Drag	DataField component to page designer - creates DataText component
Ctrl Drag	component to page designer - creates a mirror of component
Double Click	on Global Page node - loads selected page into page designer
Double Click	on Report node - actives selected report

Appendix C
Property Descriptions

Listed below is an alphabetical listing of all properties that make up the RAVE system. Properties are defined by their data type, category, components they are members of, a short description and any relationships they have with other properties. The default values are added where applicable.

AllowSplit Property

Default
False

Component/Class
Band, DataBand

Description
When set to True, the band will be allowed to split across pages.
Note:
If the band is too large to fit on a page, it will be split no matter the setting of this property.

AlwaysGenerate Property

Default
False

Component/Class
Report

Description
When set to True, generation of the complete report is forced before sending it to the output device. This is important for insuring that the report variables like TotalPages are known before the first page is printed.

Anchor Property

Default
Top / Left

Component/Class
All visible components

Description
Determines the method that will be used to align a component both vertically / horizontally within its parent's area.

See also
ExpandParent

AutoSize Property

Default

True

Component/Class

All BarCode components

Description

When set to true, BarCode dimensions will automatically resize to display all encoded text.

See also

Text

BandStyle Property

Default

'None'

Component/Class

Band, DataBand

Description

One of the powerful features of RAVE is the ability to set a band's behavior with the BandStyle property. Click on the ellipse in the BandStyle area, this will open an editor dialog window that allows the set the behavior needed for the selected band.

See also

ControllerBand, Band Style Editor

BarCodeJustify Property

Default

pjLeft

Component/Class

All BarCode components

Description

Determines where the bar code is printed relative to its bounding box.

pjLeft	Print the bar code left justified.
pjCenter	Print the bar code centered.
pjRight	Print the bar code right justified.

See also

Center, Left, Right

BarCodeRotation Property

Default

Rot0

Component/Class

All BarCode components

Description

Allows the bar code to be rotated to 4 different orientations. The pivot point for rotation is the top left corner of the bar code.

Rot0	no rotation
Rot90	rotate 90 degrees relative to page
Rot180	rotate 180 degrees relative to page
Rot270	rotate 270 degrees relative to page

See also

Left, Top

BarHeight Property

Default

0.5 (PostNet component 0.125)

Component/Class

All BarCode components

Description

Sets the height for the tallest bar.

See also

BarWidth

BarTop Property

Default

0

Component/Class

All BarCode components

Description

Sets the location of the top of the bar code. The location of the readable text is controlled by PrintReadable and PrintTop properties.

See also

PrintReadable, PrintTop, Top

Bin Property

Default

Default (Windows system settings)

Component/Class

Page

Description

Specifies the paper tray used for the document.

See also

Collate, Duplex, Orientation, PaperSize, Printer, Resolution

BorderColor Property

Default

Black

Component/Class

Circle, Ellipse, Rectangle, Square

Description

Sets the color to be used for the border of the graphic.

See also

FillColor

BorderStyle Property

Default

psSolid

Component/Class

Circle, Ellipse, Rectangle, Square

Description

Sets style of border that appears as a frame around shapes.

psClear	No border is drawn
psDash	Creates a dashed border
psDashDot	Creates an alternating dash and dot border
psDashDotDot	Creates an alternating dash and double dot border pattern
psDot	Creates a dotted border
psInsideFrame	Creates a border that is inside the frame of closed shapes
psSolid	Creates a solid border

Note:
Only psSolid can have a pen width greater than 1.

See also

BorderColor, BorderWidth

Bottom Property

Default

'None'

Component/Class

All BarCode components

Description

Sets the position for the bottom of the bar code. The value includes the readable text if it is printed.

See also

PrintReadable, PrintTop

CalcType Property

Default

ctSum

Component/Class

CalcText, CalcTotal

Description

Sets the type of calculation to be performed by the CalcText over the DataField property contents.

ctAverage	average value for a data field
ctCount	count the number of occurrences
ctMax	maximum value for a data field
ctMin	minimum value for a data field
ctSum	sum of a field

See also

CountBlanks, RunningTotal

CalcVar Property

Default

' ' (empty)

Component/Class

CalcTotal

Description

Defines the calculation component that will be used in the CalcType operation. If this property is defined, DataField will be ignored.

See also

CalcType

Categories Property

Default

' ' (empty)

Component/Class

ProjectManager

Description

Defines the available categories to which a report can belong.
For example, could define categories called 'Accounting', 'General', 'Status' and 'System'.

See also

Cursor, DevLocked, Parameters

Category Property

Default

' ' (empty)

Component/Class

Report

Description

Sets the category that a report will belong to. For example, you could define categories called 'Accounting', 'General', 'Status' and 'System'.

See also

Cursor, DevLocked, Parameters

Center Property

Default

'None'

Component/Class

All BarCode components

Description

Sets or returns the position for the horizontal center of the bar code. When a value is assigned to Center the BarCodeJustify property is set to pjCenter as well.

See also

BarCodeJustify, Left, Right

CodePage Property

Default
cpCodeA

Component/Class
I128BarCode

Description
Specifies whether Code A, Code B or Code C is being used.

cpCodeA	sets 128 output to Code A
cpCodeB	sets 128 output to Code B
cpCodeC	sets 128 output to Code C

Collate Property

Default
False

Component/Class
Report

Description
Sets the collation style of the print job for the report.

See also
Bin, Duplex, Orientation, PaperSize, Printer, Resolution

Color Property

Default
Black

Component/Class
Line and text components

Description
Used to set the color of the component's output.

See also
BorderColor, FillColor

Columns Property

Default

1

Component/Class

DataBand, Region

Description

Determines the number of columns to be used, this property is component dependent.

DataBand — Defines how many columns the band will print. The width of each column is divided evenly across the width of the band and printing progresses from the left to right column before progressing to the next row.

Region — Sets the number of columns to be use in the region. When the region is printing, it will print bands contents in all columns from left to right in a snaking fashion.

See also

Columnspacing

ColumnSpacing Property

Default

0

Component/Class

DataBand, Region

Description

Defines the width of a buffer between each column. The spacing will not be applied before the first or after the last column.

See also

Columns

ConnectionName Property

Default

' ' (empty)

Component/Class

DataView

Description

Sets the name of the data connection from which data will be retrieved for the report.

ContainsRTF Property

Default

False

Component/Class

DataMemo

Description

Indicates whether the memo contains RTF or not.

If ContainsRTF is true, then the output will be formatted according to the RTF codes contained in the memo.

Controller Property

Default

' ' (empty)

Component/Class

CalcText, CalcTotal

Description

Defines the controller component that will execute the calculation in the CalcType property when the controller is printed. Normally, the controller is tied to a particular dataview and will signal when new data is available.

See also

CalcType, ControllerBand, DataField, DataView, Initializer

ControllerBand Property

Default

' ' (empty)

Component/Class

Band, DataBand

Description

Sets the controlling band for the current Band or DataBand.

For example, a header or footer band style would assign the data of the DataBand that it is acting as a header or footer for. Detail bands in a master-detail style report would set their ControllerBand to the master DataBand.

See also

BandStyle, Controller

Copies Property

Default

1

Component/Class

Report

Description

Sets the number of copies that the report will generate when it is printed.

See also

Bin, Collate, Duplex, Orientation, PaperSize, Printer, Resolution

CountBlanks Property

Default

True

Component/Class

CalcText, CalcTotal

Description

This property sets whether the CalcText component will count blank field values for the ctAverage and ctCount calculation types.

See also

CalcType

Cursor Property

Default

crDefault

Component/Class

All drawing

Description

Will determine the type of cursor used within that item's selection zone.

crAppStart
crArrow
crCross
crDefault
crDrag
crHandPoint
crHelp
crHourGlass

See also

Categories, DevLocked, Parameters

DataField Property

Default

' ' (empty)

Component/Class

Most data aware components

Description

Defines the type of data that the component will display or process. The contents can be made up of data field names, report variables, project parameters or even string constants surrounded by quotes. It is recommended that you use the data text editor to build a DataField value.

See also

DataView

DataView Property

Default

' ' (empty)

Component/Class

Most data aware components

Description

Defines the data view that will be used if the DataField property is initialized to any field names.

See also

DataField

Description Property

Default

' ' (empty)

Component/Class

DataView, DataField, Page, Report

Description

Defines a multi-line memo that describes the component. This could be used in conjunction with the FullName property to document how the component is to be used.

See also

FullName, Name

DesignerHide Property

Default

False

Component/Class

Band , DataBand

Description

This property controls the visibility of the band contents in the designer. Reduce the "clutter" of other bands by setting their DesignerHide property to true. Then only the one(s) set to false will show. This might be needed if there are a large number of bands or a band that is occupying a large space.

Note:

This has NO effect on what will be printed, only effects what is shown on the designer page.

See also

DevLocked

DestParam Property

Default

' ' (empty)

Component/Class

CalcOp, CalcTotal

Description

value can be blank if the calculation is only going to be used as an intermediate result for other calculation components.

See also

DataField, Operator

DestPIVar Property

Default

' ' (empty)

Component/Class

Page, ProjectManager, Report

Description

Initializes the value of a PIVar (Post Initialize Variable). Any PIVars of the same name that were previously printed will show this value. PIVars are normally printed using the Data Text Editor of the DataText component. The main difference between PIVars and parameters is that PIVars will use the value that is set after (in print order) the PIVar while parameters print the value that was set before. A common use for PIVars is to print a total in a header band that would be initialized later in the footer band. This works even across multiple pages. Report.AlwaysGenerate must be true if you are using PIVars in your report.

See also

AlwaysGenerate, Editor DataText, PIVars

DetailKey Property

Default

' ' (empty)

Component/Class

DataBand , DataCycle

Description

Sets the detail key fields that will be matched to the master key fields in a master-detail report. Multiple fields should be separated with +'s.

See also

MasterKey

DevLocked Property

Default

False

Component/Class

All components

Description

Locks out the ability of anyone making changes to the property.

May be used by the developer to prevent accidental changes being made to a stable part of a complex report structure.

See also

Categories, Cursor, DesignerHide, Parameters

DisplayFormat Property

Default

' ' (empty)

Component/Class

CalcOp, CalcText, CalcTotal, All Numeric Field Types

Description

DisplayFormat formats the value given using the given format string. The various format specifiers are listed in appendix A.

See also

Appendix A

DisplayOn Property

Default

doParent

Component/Class

Most components

Description

Controls whether the component will be used during the print preview display, printer output or both.

doAll	send this item to both preview & printer
doParent	use the parent setting for DisplayOn
doPreviewOnly	item will only be displayed on the preview
doPrinterOnly	component will only be displayed on the printer

DisplayType Property

Default

dtNumericFormat

Component/Class

CalcOp, CalcText, CalcTotal

Description

Defines whether the contents represent numeric or date/time information. The DisplayFormat property will be processed as either numeric or date/time format codes depending upon this setting.

dtDateTimeFormat	sets to Date/Time format
dtNumericFormat	sets to numeric format

Duplex Property

Default

pdDefault

Component/Class

Report

Description

Will set the duplex mode for the current printer.

pdDefault	Use current mode (Duplex mode NOT changed)
pdSimplex	Simplex mode (Duplex mode NOT initialized)
pdHorizontal	Duplex mode initialized - print Head to Toe
pdVertical	Duplex mode initialized - print Head to Head

Note:
Not all printers or drivers support duplex printing.

See also

Bin, Collate, Orientation, PaperSize, Printer, Resolution

ExpandParent Property

Default

True

Component/Class

Memo

Description

When set to true, the height of the parent of the memo will be increased the same amount the height of the memo is increased.
Note:
In order to properly print dynamically sized memos; set ExpandParent to true and also set the vertical anchor to stretch.

See also

Anchor Editor, Anchor Property, DataMemo component

Extended Property

Default

False

Component/Class

Code39BarCode

Description

When set to true, Extended Code 39 format will output instead of standard Code 39 format.

FieldName Property

Default

' ' (empty)

Component/Class

All data field components

Description

Defines the field name that this field component will retrieve data from.

FileLink Property

Default

' ' (empty)

Component/Class

Bitmap, Metafile

Description

Defines a filename to initialize the bitmap or metafile with a file on the hard disk.

See also

Image

FillColor Property

Default

White

Component/Class

Circle, Ellipse, Rectangle, Square

Description

Sets the color that is used to fill the graphical shape

See also

BorderColor, Color, FillStyle

FillStyle Property

Default

fsSolid

Component/Class

Circle, Ellipse, Rectangle, Square

Description

Sets the style used to fill the graphical shape. Use the Fill Toolbar to select the fill style desired or enter it from this properties drop list box.

fsBDiagonal
fsClear
fsCross
fsDiagCross
fsFDiagonal
fsHorizontal
fsNone
fsSolid
fsVertical

Note:
Setting the FillStyle to fsClear will cause the FillColor to be set to White.

See also

BorderColor, Color, FillColor

FinishNewPage Property

Default

False

Component/Class

Band, DataBand

Description

Very similar to StartNewPage, but if set to True, it will cause a new page to begin after this band has finished printing.

Note:
The StartNewPage property would normally be used on header bands while the FinishNewPage property would normally be used on footer bands.

See also

MaxRows, StartNewPage

FirstPage Property

Default

' ' (empty)

Component/Class

Report

Description

Defines the first page that will be printed in a report if the PageList property is empty.

See also

GotoMode, GotoPage, PageList

Font Property

Default

System font

Component/Class

All text components

Description

Defines the font that will be used to draw the contents of a text component.

See also

FontJustify

FontJustify Property

Default

pjLeft

Component/Class

Most text components

Description

Sets the horizontal justification of the text data in the box.

pjBlock	block justify the text
pjLeft	left justify the text
pjCenter	center the text
pjRight	right justify the text

FontMirror Property

Default

System font

Component/Class

Most text components

Description

This property sets the FontMaster component that is used to define the font for the text. Setting this property to a font master will override the Font property.

See also

Font

FullName Property

Default

' ' (empty)

Component/Class

DataField, DataView, Page, Report

Description

Defines the full name or long name of the project item component. The full name is for single line display and may contain special characters and spaces.

See also

Description, Name

GotoMode Property

Default

gmGotoDone

Component/Class

Page

Description

This property determines the behavior of the GotoPage property.

gmGotoDone	Go to the GotoPage property setting when the page definition has completely finished printing. There may be several physical pages if there is a region defined on the page.
gmGotoNotDone	Go to the GotoPage property after a single physical page has printed, only if the current page definition has not finished printing. This option is usually only used with mirrored sections (odd/even layouts, different first page, …).
gmCallEach	Call GotoPage property after each physical page has printed whether the page definition is finished printing or not. This is useful for inserting other pages after each page of a report prints. Control is returned to the calling page when the page chain ends (i.e., a blank GotoPage property is encountered)

See also

GotoPage, PageList

GotoPage Property

Default

' ' (empty)

Component/Class

Page

Description

Defines the page that will be printed after the current page according to the GotoMode property rule.

See also

GotoMode, PageList

GridLines Property

Default

5

Component/Class

Page

Description

Controls the grid lines that are visible.

The default setting of 5 means that every fifth grid line will be visible. If GridSpacing is set to 0.1, then it means that the visible grid lines will be every 1/2 inch but snap to grid will occur every 0.1 inches.

See also

GridSpacing

GridSpacing Property

Default

0.1

Component/Class

Page

Description

Sets the spacing between grid lines.

See also

GridLines

GroupDataView Property

Default

' ' (empty)

Component/Class

Band, DataBand

Description

Defines the data view that will be used to calculate the GroupKey from.

See also

GroupKey

GroupKey Property

Default

' ' (empty)

Component/Class

Band, DataBand

Description

Defines the field(s) that will be used to calculate the group key.

When defining a report using grouping headers or footers, the GroupKey property is used to determine when a new group is encountered. Separate fields should be separated with +'s.

See also

GroupDataView

Height Property

Default

'None'

Component/Class

All visible components

Description

Defines the overall height of the component.

For bar codes this is a read only property that contains the height of the entire bar code.

If the bar code PrintReadable property is set to true, then the Height property contains the bar code height plus the line height of the current font.

See also

BarHeight, PrintReadable

HRadius Property

Default

0

Component/Class

Rectangle, Square

Description

Controls the horizontal radius of the rectangle corner. When used in combination with VRadius, these properties round the corners of rectangles or squares.

See also

VRadius

Image Property

Default

' ' (empty)

Component/Class

Bitmap, Metafile

Description

Defines the image that will be printed with a bitmap or metafile component.

See also

FileLink

InitCalcVar Property

Default

' ' (empty)

Component/Class

CalcController

Description

When this component is acting as an initializer, it defines the calculation component that will be used as the initializing value.
If this property is defined, InitDataField and InitValue will be ignored.

See also

InitDataField, InitDataView, InitValue

InitDataField {CalcController} Property

Default

' ' (empty)

Component/Class

CalcController

Description

If the InitCalcVar property is blank, then this property defines the data field or project parameter that will be used as the initial value for any component(s) using this initializer.
If this property is defined, InitValue will be ignored.

See also

InitCalcVar, InitDataView, InitValue

InitDataField {PageNumInit} Property

Default

' ' (empty)

Component/Class

PageNumInit

Description

Defines the value that the relative page number will start from when this component is printed.

If the text cannot be converted to a valid integer, a default value of 1 will be used.

See also

InitDataView, InitValue

InitDataView Property

Default

' ' (empty)

Component/Class

CalcController, PageNumInit

Description

Defines the default DataView for the InitDataField property.

See also

InitDataField

Initializer Property

Default

' ' (empty)

Component/Class

CalcText, CalcTotal

Description

Defines the initializer component. The CalcController is typically the initializer, which will initialize the calculation when the initializer is printed.

Initializers are useful for setting a calculation to a specific value or initializing it at key points in a report

See also

Controller, CalcController

InitToFirst Property

Default

True

Component/Class

DataBand, DataCycle

Description

Moves the DataSet to the Top of File or first record position.

See also

ConnectionName

InitValue {CalcController} Property

Default

0

Component/Class

CalcController

Description

If InitCalcVar and InitDataField properties are blank, then a constant value is defined that will be used as the initializing value for any components for which this is an initializer.

See also

InitCalcVar, InitDataField, InitDataView

InitValue {PageNumInit} Property

Default

1

Component/Class

PageNumInit

Description

Defines the value that the relative page number will start from when this component is printed. Only the RelativePage report variable (not CurrentPage) will reflect the new page number.

See also

InitDataField, InitDataView

KeepBodyTogether Property

Default

False

Component/Class

DataBand

Description

This property, if true, causes the data band to attempt to keep the bands from the body header to the body footer together on the same page.

See also

KeepRowTogether, OrphanRows, WidowRows

KeepRowTogether Property

Default

False

Component/Class

DataBand

Description

This property, if true, causes the data band to attempt to keep the bands from the row header to the row footer together on the same page.

See also

KeepBodyTogether, OrphanRows

Left Property

Default

'None'

Component/Class

All visible components

Description

Sets the position for the left edge of the component.

See also

Top, Width

LineStyle Property

Default

psSolid

Component/Class

All line components

Description

Sets style of line.

psClear	No line is drawn
psDash	Creates a dashed line
psDashDot	Creates an alternating dash and dot line
psDashDotDot	Creates an alternating dash and double dot line pattern
psDot	Creates a dotted pen
psInsideFrame	Creates a pen that draws a line inside the frame of closed shapes
psSolid	Creates a solid line

Note:
Only psSolid can have a pen width greater than 1 pixel.

See also

BorderStyle, BorderWidth, LineWidth

LineWidth Property

Default

1

Component/Class

Drawing

Description

Sets the width of the line. The units for this line thickness is controlled by the LineWidthType property. Line widths greater than 1 pixel can only be used with solid lines.

See also

LineStyle, LineWidthType

LineWidthType Property

Default
wtPixels

Component/Class
Drawing

Description
Determines whether the LineWidth property of the graphic shape is measure in pixels or points. Use pixels and a LineWidth of 1 (the defaults) if the thinnest line possible is needed. Also use points for all other LineWidth values for consistent thickness across devices.

WtPixels	LineWidth is measured in pixels
WtPoints	LineWidth is measured in points (1 point = 1/72nd of an inch)

See also
LineWidth

Locked Property

Default
False

Component/Class
All components

Description
Controls the state of the selected component(s) properties and all of its children (if any).
If true, then the properties cannot be selected or changed.
The color of the property names and pips (if selected) will be red when this property is true.
Note:
This can be used to a freeze part of a page design, so that the design cannot accidentally be moved while completing a design in another area.

LookupDataView Property

Default
' ' (empty)

Component/Class
DataText

Description
Specifies the data view that lookup will be performed on.

See also
DataField, LookupDisplay, LookupField, LookupInvalid

LookupDisplay Property

Default

' ' (empty)

Component/Class

DataText

Description

Specifies the field in LookupDataView that will actually be displayed in the report after the lookup is performed.

See also

DataField, LookupDataField, LookupField, LookupInvalid

LookupField Property

Default

' ' (empty)

Component/Class

DataText

Description

Specifies the field(s) in LookupDataView that will be matched to the value of the field(s) defined by DataField. Once a match is found, the value of LookupDisplay will be shown in the report.

See also

DataField, LookupDataField, LookupDisplay, LookupInvalid

LookupInvalid Property

Default

' ' (empty)

Component/Class

DataText

Description

Defines the text that will be displayed if no matches are found for the current value of DataField in the LookupDataView.

See also

DataField, LookupDataField, LookupField, LookupField

MailMergeItems Property

Default
' ' (empty)

Component/Class
All memo components

Description
Stores the tokens and replacement data text items that will be used in a search and replace during a mail merge session.

See also
Mail Merge Editor (page 129)

MasterDataView Property

Default
' ' (empty)

Component/Class
DataBand, DataCycle

Description
Defines the DataView that will be used to calculate the MasterKey property from.

See also
DetailKey, MasterKey

MasterKey Property

Default
' ' (empty)

Component/Class
DataBand, DataCycle

Description
Determines the master key field(s) for a master-detail style report. The contents of the MasterKey field(s) will be matched to the DetailKey property to set up the master-detail relationship in the detail databand. Multiple fields should be separated by +'s.

See also
DetailKey, MasterDataBand

MatchSide Property

Default

msWidth

Component/Class

Bitmap, Metafile

Description

Determines the method that the bitmap or metafile will use to resize itself.

msBoth	The image will size to match both the designed width and height.
msHeight	The image will match the designed height and adjust the width to maintain the correct proportions.
msInside	The image will be drawn proportionally inside the designed area.
msWidth	The image will match the designed width and adjust the height to maintain the correct proportions.

See also

Height, Width

MaxRows Property

Default

0

Component/Class

DataBand, DataCycle

Description

Defines the maximum number of rows that will be printed for a data band. A value of 0 means print all available rows in the data view.

See also

StartNewPage

MinHeightLeft Property

Default

0

Component/Class

Band

Description

Defines the minimum height that must remain in the region before this band will print. If the MinHeightLeft value is greater than the remaining height in the region, then band will be printed on the next page.

See also

Region

Mirror Property

Default

' ' (empty)

Component/Class

All components

Description

Will cause to currently selected component to mirror (duplicate) the properties of the component entered from the list.

See also

Global Page

Module Property

Default

None yet

Component/Class

Page

Description

This property is not implemented yet.

See also

Description, FullName

Name Property

Default

'None'

Component/Class

All components

Description

Defines the name of the component as referenced in the application's code. Use the Name property to assign a new name to the control or to find out what the name of the control is. By default, Rave assigns sequential names based on the type of the control, such as 'Rectangle1', 'Rectangle2', and so on. Change these to more meaningful names that make the code more readable. The Name must not contain any spaces or special characters. This is the name that will be used in the Project Tree Panel.

See also

Description, FullName

NullText Property

Default

' ' (empty)

Component/Class

DataField

Description

Sets the contents that will be printed if the DataField value is blank (empty).

See also

DataField, TextFalse, TextTrue

Operator Property

Default

coAdd

Component/Class

CalcOp

Description

Defines the type of operation that will be performed on the two source values. The result of this calculation can be placed in a project parameter using the DestParam property or used in other calculations as a CalcVar component.

coAdd	operation set to Source1 + Source2
coAverage	operation set to (Source1 + Source2) / 2.0
coDiv	operation set to Source1 / Source2
coExp	operation set to Source1 raised to the Source2 power
coGreater	operation set to the greater of Source1 and Source2
coLesser	operation set to the lesser of Source1 and Source2
coMod	operation set to Source1 mod Source2
coMul	operation set to Source1 * Source2
coSub	operation set to Source1 - Source2

See also

DestParam, ResultFunction, Src1Xxxxx, Src2Xxxxx

Orientation Property

Default

poDefault

Component/Class

Page

Description

Sets the orientation for a page to landscape or portrait. The value of poDefault will use the default setting for the printer to determine the orientation.

See also

Bin, Collate, Duplex, PaperSize, Printer, Resolution

OrphanRows Property

Default

0

Component/Class

DataBand

Description

Sets the minimum number of rows that can be by themselves at the bottom of a page. The default value of 0 and allows orphans (i.e. 1 row at the bottom of a page).

See also

KeepBodyTogether, KeepRowTogether, WidowRows

PageHeight Property

Default

11

Component/Class

Page

Description

Defines the height of the page. Normally this property will be modified via the PaperSize property.

See also

PageWidth, PaperSize

PageList Property

Default

' ' (empty)

Component/Class

Report

Description

Defines a list of pages to print when then report is executed. If no pages are defined in the PageList then the report will start with what is defined by the FirstPage property.

See also

FirstPage , Batch Report example on page 151

PageWidth Property

Default

' ' (empty)

Component/Class

Page

Description

Defines the width of the page. Normally this property will be modified via the PaperSize property.

See also

PageHeight, PaperSize

PaperSize Property

Default

"Letter, 8 ½ by 11 inches"

Component/Class

Page

Description

Provides a method to select the size of paper being used for a report node. Select one the report nodes, the property panel will show the properties available at the report level. Select the PaperSize and there will be a drop list where paper size can be selected.

Note:

All pages within a report must be of the same size. If they are not, the first printed page will determine the size used for all pages within that report.

See also

Orientation, PageHeight, PageWidth

Parameters Property

Default

' ' (empty)

Component/Class

Page, ProjectManager, Report

Description

Allows parameters to be defined that are available at the project, report and page level. Parameters can be used to print data that is passed from the application (e.g. ReportTitle or UserName) or to store calculation results. Parameters can be printed using the Data Text Editor of the DataText component.

See also

Editor DataText, DestParam, DestPIVar, PIVars

PIVars Property

Default

' ' (empty)

Component/Class

Page, ProjectManager, Report

Description

Allows PIVars (Post Initialize Variables) to be defined at the project, report and page level. PIVars are initialized using the DestPIVar property of the calculation components. PIVars are normally printed using the Data Text Editor of the DataText component. The main difference between PIVars and parameters is that PIVars will use the value that is set after (in print order) the PIVar while parameters print the value that was set before. A common use for PIVars is to print a total in a header band that would be initialized later in the footer band. This works even across multiple pages. Report.AlwaysGenerate must be true if you are using PIVars in your report.

See also

AlwaysGenerate, DestPIVar, Parameters, DataText

PositionMode Property

Default

pmOffset

Component/Class

Band, DataBand

Description

Determines how the position of this band will be treated relative to the previously printed band.

pmAbsolute	distances measured from top of region
pmOffset	distances measured from bottom of last band
pmOverlay	does not advance band position

See also

PositionValue

PositionValue Property

Default

0

Component/Class

Band, DataBand

Description

Sets the starting position of this band based upon the PositionMode property setting. If the setting is pmAbsolute, then this bands starting position is measured from the top of the controlling region for this band. If the setting is pmOffset, then the distance is from the bottom of the last printed band. The pmOverlay setting acts like pmOffset, however, it does not advance the "last" band printed setting.

See also

PositionMode

PrintChecksum Property

Default

'None'

Component/Class

All BarCode components

Description

Determines if the readable text includes the checksum character.

Note:

It is possible that the checksum character may not be a printable character with some of the bar code types.

See also

BarTop, UseChecksum

Printer Property

Default

' ' (empty)

Component/Class

Report

Description

Determines the printer that the report will be output to. Normally leave this field blank to print to the default printer.

See also

Bin, Collate, Duplex, Orientation, PaperSize, Resolution

PrintReadable Property

Default
True

Component/Class
All bar code components, except UPC

Description
Set this property to false, to not allow readable text to be printed along with the bar code.
Note:
For UPC bar codes, text is always printed.

See also
PrintTop, TextJustify

PrintTop Property

Default
False

Component/Class
All BarCode components

Description
Set this property to true to allow the readable text to be printed on top of the bar code. A false value means that the readable text will be printed below the bar code. This property has no effect when printing UPC codes, since the UPC text is always printed at the bottom of the bar code.

See also
PrintReadable, TextJustify

ReprintLocs Property

Default
(ALL)

Component/Class
Band, DataBand

Description
Determines whether the band reprints at the start of a new page. When a band is set to reprint on a new page, the type of band that caused the rollover to the new page will be checked against the band types in the ReprintLocs property. If they band type is found in ReprintLocs, then the band will be reprinted on the new page.

See also
BandStyle

Resolution Property

Default

prDefault

Component/Class

Report

Description

Determines the output resolution of the print job.

prDefault
prDraft
prHigh
prLow
prMedium

See also

Bin, Collate, Duplex, Orientation, PaperSize, Printer

ResultFunction Property

Default

cfNone

Component/Class

CalcOp

Description

Defines the function that will be applied to the result of the calculation after the operation has been performed.

cfAbs	result is the absolute of the value
cfArcTan	result is the ArcTan of the value in radians
cfCos	result is the cosine of the value in radians
cfDec	result is the value - 1
cfFrac	result is the decimal portion of the value
cfHoursToTime	converts the value in hours to DateTime format
cfInc	result is the value + 1
cfMinsToTime	converts the value in minutes to DateTime format
cfNeg	result is the value * -1
cfNone	result is unchanged
cfPercent	result is multiplied by 100
cfRandom	Result is a random value between 0 and the value
cfRound	Result is rounded to the nearest integer
cfRound1	Result is rounded to the 1st decimal place
cfRound2	Result is rounded to the 2nd decimal place
cfRound3	Result is rounded to the 3rd decimal place
cfRound4	Result is rounded to the 4th decimal place
cfRound5	Result is rounded to the 5th decimal place
cfSecsToTime	Converts the value in seconds to datetime format
cfSin	Result is the Sine of the value in radians
cfSqr	Result is the square of the value
cfSqrt	Result is the square root of the value
cfTimeToHours	Converts the value in datetime format to hours
cfTimeToMins	Converts the value in datetime format to minutes
cfTimeToSecs	Converts the value in datetime format to seconds
cfTrunc	Result is the integer portion of the value

See also

DestParam

Right Property

Default

'None'

Component/Class

All BarCode components

Description

Sets or returns the position for the right edge of the bar code. When a value is assigned to Right, the BarCodeJustify property is set to pjRight as well.

See also

BarCodeJustify, Center, Left

Rotation Property

Default

0

Component/Class

CalcText, DataText, Text

Description

Defines the rotation in degrees of the selected text component. The text is rotated counter clockwise through the values 0 to 359.

See also

BarCodeRotation

RunningTotal Property

Default

False

Component/Class

CalcText, CalcTotal

Description

Determines if a running total is kept when a NewPage occurs.

False	means NO and the value is reset to 0 each time it is printed
True	means Yes and the value is kept and will continue to total after it is printed

See also

CalcType, CountBlanks

Size Property

Default

'None'

Component/Class

DataField

Description

Defines the character width of the data field. This is normally used with design time activities such as dragging and dropping DataText components to determine an approximate width that it will take to print the component.

SortKey Property

Default

' ' (empty)

Component/Class

DataBand, DataCycle

Description

Defines the field(s) that will be passed to the data connection to set up a sort. Separate multiple fields with +'s.

Note:

The underlying database must support the fields being passed as a sort key.

Src1CalcVar Property

Default

' ' (empty)

Component/Class

CalcOp

Description

Defines a calculation component to use as the first source value for the calculation operation. If this property is defined, Src#DataField and Src#Value will be ignored.

See also

Src#DataField, Src#DataView, Src#Function, Src#Value

Src1DataField Property

Default

' ' (empty)

Component/Class

CalcOp

Description

If Src#CalcVar is blank, defines a data field to use as the first source value for the calculation operation. If this property is defined, Src#Value will be ignored.

See also

Src#CalcVar, Src#DataView, Src#Function, Src#Value

Src1DataView Property

Default

' ' (empty)

Component/Class

CalcOp

Description

Defines the default dataview that will be used for the Src#DataField property.

See also

Src#DataField

Src1Function Property

Default

cfNone

Component/Class

CalcOp

Description

Defines the function that will be performed on the source value before the calculation operation is performed.

Note:

see ResultFunction (for list of constants).

See also

Src#CalcVar , Src#DataField , Src#Value

Src1Value Property

Default

0

Component/Class

CalcOp

Description

If Src#CalcVar and Src#DataField are blank, defines a constant value that will be used as the first source value for the calculation operation.

See also

Src#CalcVar, Src#DataField, Src#Function

Src2CalcVar Property

Default

' ' (empty)

Component/Class

CalcOp

Description

Defines a calculation component to use as the first source value for the calculation operation. If this property is defined, Src#DataField and Src#Value will be ignored.

See also

Src#DataField, Src#DataView, Src#Function, Src#Value

Src2DataField Property

Default

' ' (empty)

Component/Class

CalcOp

Description

If Src#CalcVar is blank, defines a data field to use as the first source value for the calculation operation. If this property is defined, Src#Value will be ignored.

See also

Src#CalcVar, Src#DataView, Src#Function, Src#Value

Src2DataView Property

Default

' ' (empty)

Component/Class

CalcOp

Description

Defines the default dataview that will be used for the Src#DataField property.

See also

Src#DataField

Src2Function Property

Default

cfNone

Component/Class

CalcOp

Description

Defines the function that will be performed on the source value before the calculation operation is performed.

Note:

see ResultFunction (for list of constants).

See also

Src#CalcVar , Src#DataField , Src#Value

Src2Value Property

Default

0

Component/Class

CalcOp

Description

If Src#CalcVar and Src#DataField are blank, defines a constant value that will be used as the first source value for the calculation operation.

See also

Src#CalcVar, Src#DataField, Src#Function

Tag Property

Default

nil

Component/Class

All components

Description

Tag has no predefined meaning to Rave. The Tag property is provided for the convenience of storing additional integer value or pointer information for special needs in an application. For example, use the Tag property when implementing case statements with a component.

Text Property

Default

'None'

Component/Class

Text components and all bar code components

Description

For bar codes, do not include the check character. The check character will be automatically calculated and printed according to the state of the UseChecksum property.

Note:

For bar codes, any characters that are invalid for the bar code type will be deleted from the text property upon assignment.

See also

Font , FontJustify

TextFalse Property

Default

' ' (empty)

Component/Class

BooleanField

Description

Determines what will printed is the field value is False. A blank value for this property will print the text "False".

See also

TextTrue

TextJustify Property

Default

pjCenter

Component/Class

All BarCode components

Description

Determines how the readable text is justified in relation to the bar code.

pjBock	Block justify the text portion
pjLeft	Left justify the text portion
pjCenter	Center justify the text portion
pjRight	Right justify the text portion

See also

PrintReadable, PrintTop, Text

TextTrue Property

Default

' ' (empty)

Component/Class

BooleanField

Description

Determines what will printed is the field value is True. A blank value for this property will print the text "True".

See also

TextFalse

Top Property

Default

'None'

Component/Class

All visible components

Description

Sets or returns the position for the top edge of the component. For bar codes, the value for this property includes the readable text, if it is printed.

See also

BarTop, Left, PrintReadable, PrintTop

Truncate Property

Default

False for CalcText and Text, True for DataText

Component/Class

CalcText, DataText, Text

Description

When set to true, the text will be truncated to fit the Width property defined in the designer for that component. If the property is false, then all of the text will print without regard to the width property.

See also

Width

Units Property

Default
unInch

Component/Class
Project

Description
Sets the default units mode for this project. If the setting is unUser then units factor is determined by the value in UnitsFactor.

unCM	Units are in centimeters
unInch	Units are in inches
unMM	Units are in millimeters
unPoint	Units are in points (1/72nd inch)
unUser	Unit are custom defined in UnitsFactor

See also
UnitsFactor

UnitsFactor Property

Default
1.0

Component/Class
Project

Description
Sets or returns the current conversion factor necessary to convert units to inches. Its value should equal the number of units that equal an inch. (unCM = 2.54 since 2.54 centimeters equal an inch)

See also
Units

UseCheckSum Property

Default
False (Code128 := true)

Component/Class
All BarCode components

Description
Specifies whether a checksum character should be included in the bar code.

See also
BarHeight, BarWidth, PrintReadable, Text, Width

VRadius Property

Default

0

Component/Class

Rectangle , Square

Description

Controls the vertical radius of the rectangle corner. When used in combination with HRadius, these properties round the corners of rectangles or squares.

See also

HRadius

WasteFit Property

Default

True

Component/Class

Page, Section

Description

If true, components on the page or within the section will dynamically adjust themselves to fit within the printer's waste margins. If a component is not located within the waste area, no adjustment will occur unless it's parent component is located in the waste area.

It is important that components (within a page or section) use the "Anchor" property so that they will adjust as the margins change.

See also

Anchor, Anchor Editor, PageHeight, PageWidth, PaperSize

WideFactor Property

Default

3.0

Component/Class

All BarCode components

Description

The wide factor is the ratio of the wide bar to the narrow bar width.

See also

BarHeight, BarWidth, Width

WidowRows Property

Default

0

Component/Class

DataBand

Description

Sets the minimum number of rows that can be by themselves on the top of the next page. The default setting is 0 and allows widows.

See also

KeepBodyTogether, KeepRowTogether, OrphanRows

Width Property

Default

'None'

Component/Class

All components

Description

Sets the width of the component. For bar codes, this will return the calculated width of the entire bar code for the current value of Text.

See also

BarWidth, Height, Left, Text, WideFactor

WidthType Property

Default

wtPixels

Component/Class

Drawing components

Description

Determines how the width of lines is calculated. Generally *wtPixels* and a line width of 1 pixel should be used when the thinnest possible line is desired (typically referred to as hairline). The width type of *wtPoints* should be used on all line widths greater than one pixel to maintain consistency between devices of different DPI resolutions (i.e. Printer and Preview). When using *wtPoints*, line widths are expressed in terms of points (72 points per inch).**See also**

BorderWidth, LineWidth

INDEX

Borland®

What's New in Borland® Delphi™ 7 Studio

Improve developer productivity and enhance performance

by Technical Publications
Borland Software Corp.
August 2002

Contents

Introduction

Borland® Delphi™ 7 Studio includes new features and enhancements in the following areas:

- IDE
- Web technology
- COM technology
- Database technology
- Component library
- Runtime library
- Compiler
- Support for Rave Reports
- Support for ModelMaker
- Documentation

If you are upgrading from a previous version of Delphi, see "Upgrade and compatibility issues" on page 7.

IDE

The IDE has new features in the following areas:

Compiler messages

- The new View|Additional Message Info command displays a Message Hints window from which you can download and view information about compiler messages from the Borland Web site.
- The new Project|Options|Compiler Messages page gives you greater control over which compiler warnings are generated.

Component palette

- When you open a Borland CLX™ (Component Library for Cross-platform) application in Delphi, a new CLX-only version of the System page is displayed. It includes several directory and file components. In previous releases, the System page was displayed only for VCL applications and included components for system-level access.
- The new Indy Intercepts and Indy I/O Handlers pages provide open source Internet protocol components (Professional and Enterprise editions).
- The new IW Standard, IW Data, IW Client Side, and IW Control pages provide AtoZed Software IntraWeb components for developing Web-based applications.
- The new Rave page provides components for adding report generation to your applications.
- If a component page can be scrolled horizontally to display additional icons, a new drop-down menu button can also be used to list the additional icons.

CodeInsight™

- Code completion is now faster and lets you browse to the declaration of items in the code completion list by using Ctrl+click on any identifier in the list.
- New HTML code completion automatically displays valid HTML elements and attributes in the Code editor (Professional and Enterprise editions only).
- You can create customized code completion managers by using the OpenTools API. See “Extending the IDE” in the Delphi online help for details.
- The Tools|Editor Options|Code Insight page lets you set colors for the symbols displayed in the CodeInsight™ tools.

Debugger

- The Watch List now has:
    - Multiple tabs, allowing you to organize watches into distinct watch groups for easier debugging. To add a watch group, right-click the Watch List and select Add Group.
    - A Watch Name column and a Value column. To show/hide the column headers, right-click the Watch List and select Show Column Headers.
    - A checkbox to enable or disable individual watches.
- The Tools|Debugger Options|Event Log page has the following new options:
    - Use Event Log Colors lets you display different types of event messages in color in the event log.
    - Module messages writes a message to the event log each time a module (exe, dll, ocx, etc.) is loaded or unloaded by the process that you are debugging. Previously, the Process messages option controlled whether these events were logged.
- The Run Parameters dialog box has a new option, Working Directory, that lets you specify the name of the directory to use for the debugging process.

Miscellaneous IDE improvements

- From the Project Manager, you can partially compile projects within a group by right-clicking on any project and choosing Make All from Here or Build All from Here.
- The Message view has multiple tabs for displaying different types of messages (Build, Search, and so on).
- The View|Component List command lets you multiselect components by pressing the Ctrl key.

- The new Tools|Editor Options|Source Options page lets you:
    - Set different editor options for different source types, such as Pascal, C++, C#, HTML, and XML.
    - Display tab and space characters in the Code editor.
    - Edit code templates.
    - Several of the options on the new page were formerly on the General, Display, and CodeInsight pages of the Editor Properties dialog box.
- The Tools|Editor Options|Color page has two new options, Foreground Color and Background Color, instead of a color grid, for setting colors in the Code Editor.
- Pressing Alt+Page Down and Alt+Page Up cycles through tabbed views such as the Code Editor, Watch Window, and Message view. These keyboard shortcuts are included in the Default, IDE Classic, and BRIEF key mappings.
- Delphi now displays a two-tone main menu.

Web technology

In Delphi 7 Studio Enterprise and Professional editions:

- Delphi now includes IntraWeb from AToZed Software. You can use IntraWeb to develop Web server applications using standard form tools. You can also use IntraWeb to develop pages for Borland WebBroker™ and WebSnap™ applications. For more information, see "Creating Web server applications using IntraWeb" in the *Developer's Guide* or online Help. Delphi 7 Studio Enterprise includes the complete IntraWeb product. Delphi 7 Studio Professional includes a subset of the IntraWeb product.
- Delphi now supports Apache™ 2 as a target type for WebBroker, WebSnap, and SOAP.
- Borland has deprecated Win-CGI as a target type for Web server applications and Web Services. Borland recommends using regular CGI, ISAPI/NSAPI, or an Apache target type instead. Existing Win-CGI projects can still be modified and compiled in the IDE, however, Borland does not guarantee Win-CGI compatibility for the indefinite future.

Web Services

Web Services includes the following enhancements.

New UDDI browser

The WSDL Import Wizard has a new Universal Description, Discovery, and Integration (UDDI) browser that lets you search a UDDI registry for a Web Service and import the address of its WSDL document.

SOAP headers

New classes and interfaces let you read or insert headers into the SOAP envelopes that transmit messages between clients and servers. For more information, see "Defining and using SOAP headers" and "Processing headers in client applications" in the *Developer's Guide* or online Help.

Attachments

Web Services applications (both client and server) can now handle attachments. Attachments (TSOAPAttachment descendants) are sent with SOAP-encoded messages as part of a multipart form. When an application receives the attachment, it saves it to a temporary file, which is then available to your application.

Type support

- You can now customize the conversion between remotable classes and their SOAP representation by overriding two new virtual methods that were added to TRemotable: ObjectToSOAP and SOAPToObject.
- Exception objects for exceptions that occur when responding to a Web Service request (ERemotableException instances) now contain more information from the SOAP fault packet.
- Type definitions are automatically registered with the remotable type registry when you register an invokable interface.

- TXSDecimal has a new AsBcd property for easier conversion between XML and native types. Similarly, TXSHexBinary has a new AsByteArray property. Remotable classes that represent time values now let you work with fractional seconds rather than milliseconds.

Other enhancements

- New events on THTTPReqResp let you to intercept the HTTP message before it is sent and to monitor progress while sending or receiving long messages.
- THTTPSoapPascalInvoker now publishes events that let you write code to execute before or after the invoker executes a requested method call.
- You now have more control over the mapping between invokable interfaces and WSDL documents. TWSDLHTMLPublish now publishes several events to let you control the generated WSDL. You can also identify the mapping between function return values and parameter names, the use of namespaces, and default SOAP actions. On the client side, literal encodings are now supported as well as RPC-style encoding.
- A new interface, IRIOAccess lets you access the remote interfaced object that implements an invokable interface.
- The IOPConvert interface has a new property: Encoding. This allows you to specify the character set to use for encoded messages that are passed between the client and Web Services provider.
- There are changes to Web Services that affect Borland DataSnap™ applications. For more information, see "Database technology " on page 4.
- The TLinkedRIO constructor now automatically generates separate file names for each method you call, making debugging easier.
- TOPToSoapDomConvert now has two new events that you can use when debugging the deserialization of SOAP packets.
- You can now use overloaded methods on invokable interfaces that you define.

COM technology

In Delphi 7 Studio Enterprise and Professional editions:

You can now use the Import Type Library dialog box (Project|Import Type Library) to create a CoClass wrapper for Microsoft® .NET assemblies. You can use the resulting wrapper as you would an ordinary COM server, using the interoperability features of .NET.

Database technology

In Delphi 7 Studio Enterprise and Professional editions:

- The dbExpress drivers have been updated for IBM® DB2® 7.2 and Informix® SE, Oracle9i,™ Borland InterBase® 6.5, and MySQL™ 3.23.49. A new driver is available for Microsoft SQL 2000.
- There are several new and changed database components. See "Component library " on page 5 for details.
- Borland has deprecated SQL Links; no further enhancements will be made to SQL Links, and it will not be included with Delphi after 2002. Borland recommends using dbExpress for SQL server database access in Delphi.

DataSnap™

In Delphi 7 Studio Enterprise edition only:

- In DataSnap applications, the use of IAppServer has been changed to IAppServerSOAP, which avoids some ambiguities in the IAppServer interface. The UseSOAPAdapter property of TSoapConnection can be used to write clients for servers written with earlier versions of Delphi. TSoapConnection also publishes several new events for you to customize your client applications at various points in the process of executing a Web Services request.
- You can now identify a specific SOAP data module in an application server that has multiple data modules. Use the SOAPServerIID property or add the data module's interface to the end of the URL.

- You can now use the SOAP connection component to call extensions to the application server's interface. Use the SOAPServerIID property and the GetSOAPServer method.
- DataSnap no longer supports CORBA® connections.

Component library

Support for Windows XP™ Themes

In Delphi Studio 7 Enterprise and Professional editions, Borland VCL applications now include components that enable support for Windows® common controls version 6. Your application will automatically use the new Windows controls on Windows XP™ systems if it finds a suitable manifest file. For more information, see "Common controls and XP themes" in the *Developer's Guide* or online Help.

New unit

The new DBClientActns unit contains three new action components for working with client datasets: TClientDataSetApply, TClientDataSetUndo, and TClientDataSetRevert.

New components

- The dbExpress page of the Component palette includes TSimpleDataSet for use with simple, two-tier database applications (TSimpleDataSet replaces TSQLClientDataSet).
- The Dialogs page of the Component palette includes TPageSetupDialog for providing a Windows-standard page setup dialog box.
- The Additional page of the Component palette includes TXPColorMap, TStandardColorMap, and TTwilightColorMap for colorizing menus and toolbars.
- The new CLX version of the System page of the Component palette includes new directory and file components.
- The new Indy Intercepts and Indy I/O Handlers pages on Component palette provide Internet protocols in Professional and Enterprise editions.

Changed components

- The CLX versions of TOpenDialog and TSaveDialog have been expanded to support additional features such as file previewing.
- The VCL version of TCustomForm has two new properties, ScreenSnap and SnapBuffer, which control whether a form snaps to the edge of the screen when the form is moved.
- TCustomComboBoxEx has a new AutoCompleteOptions property that enables a combo box to respond to user keystrokes.
- CLX dialog objects that descend from TOpenDialog and TQtDialog can now use Windows Common Dialogs in place of Qt Dialogs. This behavior is controlled by the UseNativeDialog property, which defaults to true.

Deprecated components

Information about deprecated components can be found in the readme.txt file in the Delphi 7 Studio directory.

Runtime library

Classes unit

- A new exception class, EFileStreamError, has been added. EFileStreamError and EFOpenError descend from this class. This new class may take a FileName parameter. As a result, the exception message text now contains the name of the file the error occurred on.
- The TStrings class has two new properties, ValueFromIndex and NameValueSeparator.
- The TThread.CheckThreadError methods have been promoted from private to protected visibility.

Math unit

The Math unit has a new default parameter, RaisePending, in the ClearExceptions procedure.

StdConvs unit

The StdConvs unit now includes stones in the supported weight units.

StrUtils unit

The StrUtils unit contains the following changes related to multi-byte character set (MBCS) support:

- Previously, LeftStr, RightStr, and MidStr each had an AnsiString parameter type and return type, and did not support MBCS strings. Each of these functions has been replaced by a pair of overloaded functions, one that takes and returns AnsiString, and one that takes and returns WideString. The new functions correctly handle MBCS strings. This change breaks code that uses these functions to store and retrieve byte values in AnsiStrings. Such code should be updated to use the new byte-level functions described below.
- New functions LeftBStr, RightBStr, and MidBStr provide the byte-level manipulation previously provided by LeftStr, RightStr, and MidStr.
- New functions AnsiLeftStr, AnsiRightStr, and AnsiMidStr are the same as the new AnsiStr LeftStr, RightStr, and MidStr functions, except that they are not overloaded with equivalent WideString functions.

The StrUtils unit has a new string-searching function called PosEx.

SysUtils unit

The SysUtils unit now includes thread-safe overloads of routines that format and parse numbers, date-time values, and currency. The new routines are thread-safe because they obtain their localization information from a TFormatSettings data structure instead of from global variables. This data structure must be populated before being used; a new function, GetLocaleFormatSettings, is provided to populate the data structure from a specified locale.

VarCmplx unit

The VarCmplx unit has new functions: VarComplexLog2, VarComplexLog10, VarComplexLogN, VarComplexTimesImaginary, and VarComplexTimesReal.

Variants unit

- The VarIsError and VarAsError functions have been added.
- The EVariantError exception is now a base class for finer grained exception classes that are thrown from variants code.
- Several new global Variant control variables have been added: NullEqualityRule, NullMagnitudeRule, NullStrictConvert, NullAsStringValue, and PackVarCreation.

Compiler

The Delphi compiler now supports three additional compiler warnings: Unsafe_Type, Unsafe_Code, and Unsafe_Cast. These warnings are disabled by default, but can be enabled with the compiler directive `{$WARN UNSAFE_CODE ON}`, compiler command line switch `dcc32 -W+UNSAFE_CODE`, and, in the IDE, on the Project|Options|Compiler Messages page.

This feature is intended to help you port your code to the managed execution environment of the .NET platform. In a managed execution environment, "unsafe" means the operation cannot be verified during the static analysis performed by the Just In Time (JIT) compiler. Such code might pose a security risk, since there is not enough information for the JIT compiler to verify its runtime behavior. Examples of unsafe code include pointer operations and memory overwrites.

Support for Rave Reports

In Delphi 7 Studio Enterprise and Professional editions:

The Delphi environment now includes Rave Reports from Nevrona. By adding Rave Reports components to your application, you can enable your users to generate reports within your application. For more information see “Creating reports with Rave Reports” in the *Developer’s Guide* or online Help.

Support for ModelMaker

In Delphi 7 Studio Enterprise and Professional editions: ModelMaker tools can help simplify the design, construction, and maintenance of classes and interfaces. ModelMaker also includes tools for creating UML™-style diagrams, which can be used to create and modify your projects' source code. For more information, see "Designing classes and components with ModelMaker" in the *Developer's Guide* or online Help.

Delphi 7 Studio Enterprise includes ModelMaker from ModelMaker software. Delphi 7 Studio Professional includes a 30-day trial version of ModelMaker. The ModelMaker functionality is the same in both editions of the Delphi environment.

Documentation

- All of the documentation files (PDF, HTML, and INT) are now distributed in the **Online+PDF Docs** folder on the Delphi Companion Tools CD instead of the installation CD. You can access the documentation directly from the CD or copy it to the folder of your choice.
- Due to size constraints for the printed *Developer's Guide*, Part V "Creating custom components," has been removed from that book to create the new *Component Writer's Guide.* The new book is available in the online Help and as a PDF file on the Delphi Companion Tools CD.
- The Object Pascal language is now called the Delphi™ language. The online Help and documentation have been updated accordingly.
- The *Object Pascal Language Guide* is now the *Delphi Language Guide*.
- To ensure the continued accuracy of the Delphi tutorials, they have been removed from the *Quick Start* and the *Developer's Guide*. The tutorials are available as PDF files on the Delphi Companion Tools CD.
- Some of the Delphi online help topics include C++ syntax and code examples for Borland Kylix™ and C++Builder™ users. For Delphi development, please disregard these examples.

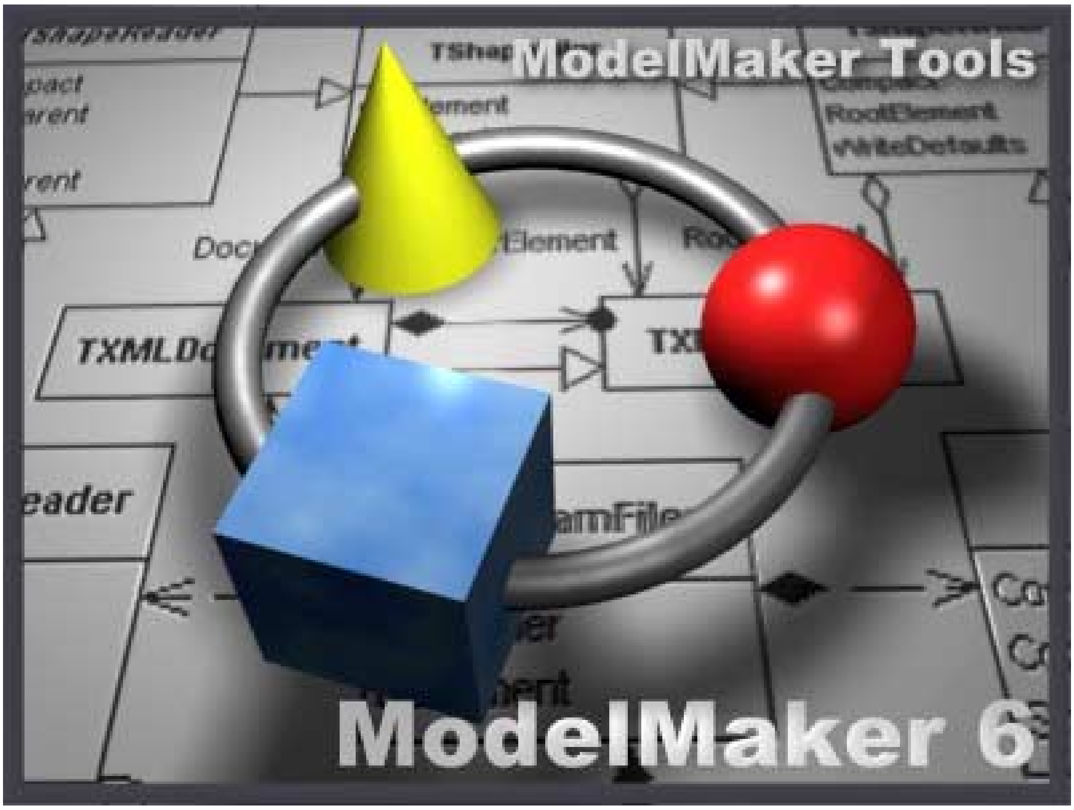
Upgrade and compatibility issues

For late-breaking upgrade and compatibility issues, see the readme.txt file in the Delphi 7 installation directory.

- To upgrade a Delphi (formerly Object Pascal) language project from a previous version of Delphi, open it in the new version. The project is automatically updated to the new release.
- Details in Fault messages are now added to the <detail> node rather than as children of the <detail> node. This brings our handling of SOAP faults into accordance with the SOAP specification, but breaks backward compatibility with older code.
- Changes to the StrUtils unit LeftStr, RightStr, and MidStr functions may require you to update code that uses these functions. See "Runtime library " on page 5 for details.
- For Apache 2, the variable "ContentType" has been changed to "handler" in the ApacheApp unit.
- DataSnap no longer supports CORBA® connections.

Borland

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000



User Manual
ModelMaker 6.20

ModelMaker Tools
Stenenkruis 27 B
6862 XG Oosterbeek
The Netherlands

http:\\www.modelmakertools.com
info@modelmakertools.com

http:\\www.modelmaker.demon.nl
info@modelmaker.demon.nl

ModelMaker Tools
Stenenkruis 27 B
6862 XG Oosterbeek
The Netherlands

http:\\www.modelmakertools.com
info@modelmakertools.com

http:\\www.modelmaker.demon.nl
info@modelmaker.demon.nl

This user manual focuses on essentials and how things are done in ModelMaker. A GUI reference is available as context sensitive help file. This contains the latest GUI details. The Design Patterns manual focuses on ModelMaker's Design patterns and contains another step by step demo.

Author: G. Beuze

Contents

Introduction

ModelMaker represents a brand new way to develop classes and component packages for Borland Delphi 1-6. ModelMaker is a two-way class tree oriented productivity**,** refactoring and UML-style CASE tool specifically designed for generating native Delphi code (in fact it was made using Delphi and ModelMaker). Delphi's Object Pascal language is fully supported by ModelMaker. From the start ModelMaker was designed to be a smart and highly productive tool. It has been used to create classes for both real-time / technical and database type applications. ModelMaker has full reverse engineering capabilities.

ModelMaker supports drawing a set of UML diagrams and from that perspective it looks much like a traditional CASE tool. The key to ModelMaker's magic, speed and power however is the active modeling engine which stores and maintains all relationships between classes and their members. Renaming a class or changing its ancestor will immediately propagate to the automatically generated source code. Tasks like overriding methods, adding events, properties and access methods are reduced to selecting and clicking.

The main difference between ModelMaker and other CASE tools is that design is strictly related to and native expressed in Delphi code. This way there is a seamless transition from design to implementation currently not found in any other CASE tool. This approach makes sure your designs remain down to earth. The main difference between ModelMaker and other Delphi code generators are it's high level overview and restructuring capabilities letting you deal with complex designs.

A unique feature, currently not found in any development environment for Delphi, is the support for design patterns. A number of patterns are implemented as 'ready to use' active agents. A ModelMaker Pattern will not only insert Delphi style code fragments to implement a specific pattern, but it also stays 'alive' to update this code to reflect any changes made to the design

As a result, ModelMaker lets you:

- Speed up development
- Produce designs and code of unequaled quality.
- Think of designing code instead of typing code.
- Design without compromising.
- Refine and experiment with your designs until they are just right.
- Create and maintain magnitudes larger models in magnitudes less time.
- Document you designs in UML style diagrams.
- Document your units in help files by clicking a single button.

- In short: *save time and money, making better software.*

Installation

For installation details, refer to the readme.txt which is part of all ModelMaker distribution archives. We suggest you read this file before installing. The readme.txt also contains the latest information available on precautions related to upgrading.

ModelMaker requires Windows 95/98/ME/2000 or Windows NT 4.0 Both Borland Delphi and ModelMaker use a lot of resources. This might lead to problems under resource limited systems as Win95/98/ME.

ModelMaker is designed to work on a high resolution monitor (800x600 or better).

Contacting ModelMaker Tools

We at find it important to support you in your use of ModelMaker all the ways we can. You can find ModelMaker on the Internet at http:\\www.modelmakertools.com or alternatively http:\\www.modelmaker.demon.nl At this web site:

- We’ll update you on the most recent news concerning ModelMaker and it’s development.
- We’ll have the latest demo versions available.
- You can consult the Tips, FAQ pages.
- You’ll find links to the web-based ModelMaker newsgroups.
- You can leave hints or requests for future versions of ModelMaker.
- You can report bugs.

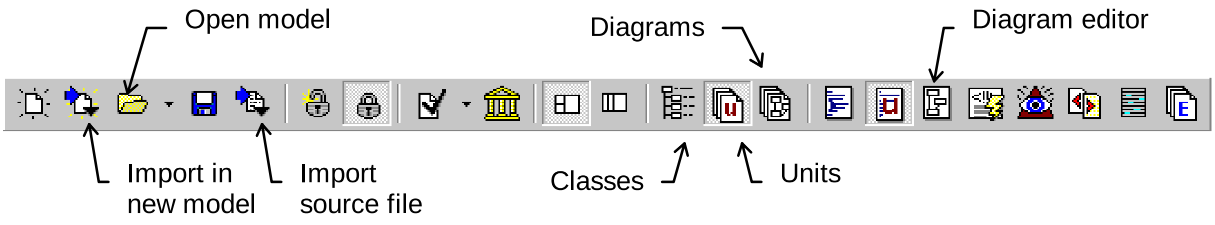
All ‘how to do this’ are best asked in the ModelMaker newsgroups. For other questions, the address to contact us is: info@modelmakertools.com or info@modelmaker.demon.nl.

Getting started

Getting a first impression

Here are some examples to get you started without reading lots of text. The development model will be explained in detail in the next chapters.

Loading an example model



To get a first impression of what ModelMaker is capable of, load the model ..\ModelMaker\6.0\Demos\MMToolsApi.mpb. It contains the interfaces making up ModelMaker's open tools API. In diagrams the relations between the interfaces defining this API are visualized. The Classes view shows the inheritance relations of this unit.

Then, to see how ModelMaker treats classes and units, use the 'Import source file in new Model' command from the toolbar. This will create a new model and import a Delphi unit (such as a form unit or a VCL unit). The model is named after the unit: Importing unit1.pas will create model unit1.mpb. Note the use of source aliases in the popup menu associated with the tool button. Source aliases are used to locate the source file and will be explained in chapter Source Aliases, page 52

Visualizing existing code

Visualizing existing code is a also good way too of getting started with ModelMaker. To visualize code:

1. Import the units containing the classes to visualize. Use the 'import source file" tool button in the main toolbar or drag drop source files on the 'unit's view' (View|Units)
2. Create or select a new class diagram in the 'diagrams' view (View|Diagrams)
3. In the Diagram editor (View|Diagram Editor) select the visualization wizard from the Wizard popup-up sub-menu.
4. Use this wizard to select the classes and interfaces to visualize and the kind of relations to visualize (inheritance, uses, supports etc.)
5. Completing the wizard gives you an instant diagram of the code just imported.

You might want to move around classes or interfaces (Drag move) or select different display options for classes or interfaces (Double click on the symbol) or the diagram as a whole

(Double click in empty space). Try to turn on and off member display, select interface style etc.

Creating code with ModelMaker, overview

A ModelMaker model contains a Code Model and Diagrams. The Code Model contains the classes, class members (properties, methods), units etc. that map to the corresponding entities in Delphi's Object Pascal. Diagrams are used to visualize aspects of the code model or entities that do not exists in the code model at all such as use cases. This 'Getting started' example will focus on the code model and demonstrate creating a new unit containing a new component class.

To create code for a new (component) class (or interface) in ModelMaker you will at a minimum need to,

1. Create a new model in which you want to put related classes, if you don't want to add the new class to the current model.
2. Add a new class to the model defining it's class name and ancestor.
3. Add (or override) properties, methods and events to the class's interface.
4. Implement the new methods.
5. Add the new class to a (new) unit.
6. Generate the unit to actually create or update a source file on disk.
7. In Delphi, debug the unit, and if it contains components, add it to the VCL.
8. While debugging, keep editing your code in ModelMaker, switching between Delphi and ModelMaker using ModelMaker's integration experts

An alternative way is to import existing files into a new model to either maintain these units in ModelMaker or to derive new classes from. This will be explained in detail in the Import demo.

The more advanced features of ModelMaker demonstrated in this example include:

9. Creating documentation for your component.
10. Generating a Help file.
11. Creating a class diagram to document your design.

The demo component: TIntLabel

Let's examine these steps a little closer by creating an new component class `TIntLabel` which is a `TLabel` descendant. `TIntLabel` adds a property `NumValue` of type integer which simple converts the `Caption` property to an Integer. We'll store this class in a new file `INTLABEL.PAS` and register it on page '*MM Demo*' in the VCL. We'll also create the help file `INTLABEL.HLP` and integrate it with Delphi's on line help.

This demo project is also shipped with ModelMaker. You can load the `GETSTART.mpb` in the `[installdir]\DEMOS` folder. The source file `INTLABEL.PAS` is also in this folder.

The ModelMaker Class Creation Wizard

If you start up ModelMaker the first time you'll see the Class Creation Wizard. This wizard makes it easy to create new classes and add them to a (new) unit. This wizard can be found at the main menu 'Tools|Create Class wizard'.

The wizard is great for adding classes, but for demonstrating the ModelMaker development model it's more instructive to create a new class manually. Therefore, if you started ModelMaker and the wizard is automatically started, abort the wizard by clicking 'Cancel'. You might also want to uncheck the option 'Show at start up' which will stop the wizard from appearing each time you run ModelMaker.

Creating a new project

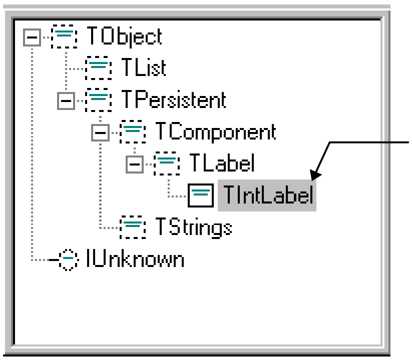
There are three ways to create a new project (or model):

- Select 'File|New', you'll get a clean project just containing the default ancestors `TObject` and `IUnknown`.
- Select 'File|New from default', you'll get a new project loaded from the default template.
- Select 'File|New from template', you'll select a template other than the default to create a new project.

In this case select 'File|New from default' to create a project which at least contains the `TComponent` class, if you didn't modify the default project shipped with ModelMaker `[installdir]\BIN\DEFAULT.mpb`.

Creating new classes

We'll use the Classes view to create a new class. In this view you add a new class as a descendant to another class in the model. The Classes view is depicted here.



Class tree containing place holder TLabel and real class TIntLabel

The ancestor class must *always* be part of the model since ModelMaker needs it to correctly generate the class declaration. In our case this implicates that before adding the `TIntLabel` class, it's ancestor `TLabel` must exist in the model. This raises a problem. If you started with the same default model, or with a template model that did not contain the class

`TLabel`, you will have to add the `TLabel` class first. But, in order to correctly add a class `TLabel` to your model, you now need to add it's ancestor first, and before that, etc.... help!

Since you do not intend to create code for `TLabel`, but only use it as an ancestor, there is no need to have the correct ancestor for `TLabel`. In our example the ancestor for `TLabel` could be anything, for example `TObject`. A better fitting ancestor is of course `TComponent`. Classes like `TLabel` in our example are called 'placeholder' classes as opposed to 'real' classes such as `TIntLabel`. Other examples of placeholders are `TObject`, Delphi's default class ancestor and `TComponent`.

What we have to do now, is add two classes `TLabel` ('placeholder') and then add `TIntLabel` ('real' class).

To do so:

1. Make the Classes view visible by selecting 'View|Classes' (or press F3)
2. Select `TComponent` by clicking it.
3. Press the "Ins" key or select add 'Add descendant' from the popup menu.
4. Enter `TLabel` as class name. You might want double click the class and in the class editor dialog check the option 'placeholder' to make it clear that `TLabel` is just a substitute for the real `TLabel` (which is in unit `StdCtrls`).
5. Now add the class `TIntLabel` using the `TLabel` as it's ancestor the same way. Of course you don't check 'placeholder' here.

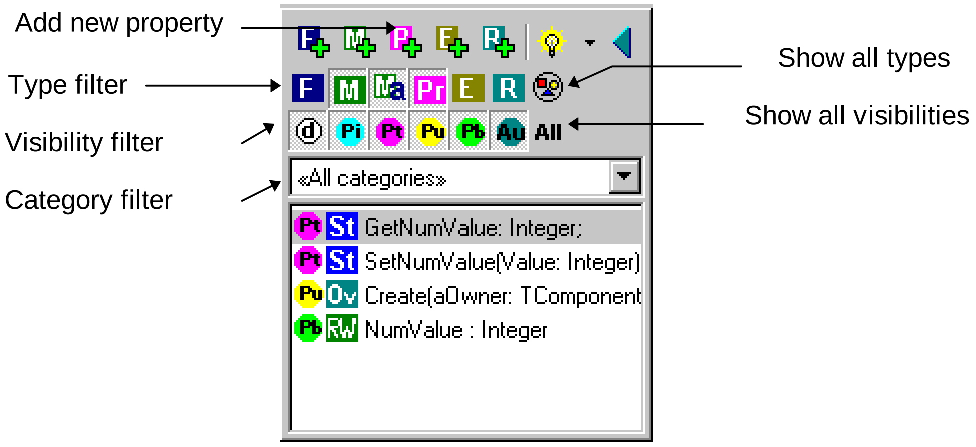
In the Classes view you'll see a tree or list based overview of all classes (and interfaces) in the model. Use the popup menu to toggle between tree and list style.

Adding properties and methods to a class

In our example we now need to add a new property and a read and write access method to the interface of the class `TIntLabel`, to get something like:

```
type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue: Integer read GetNumValue write SetNumValue;
  end;
```

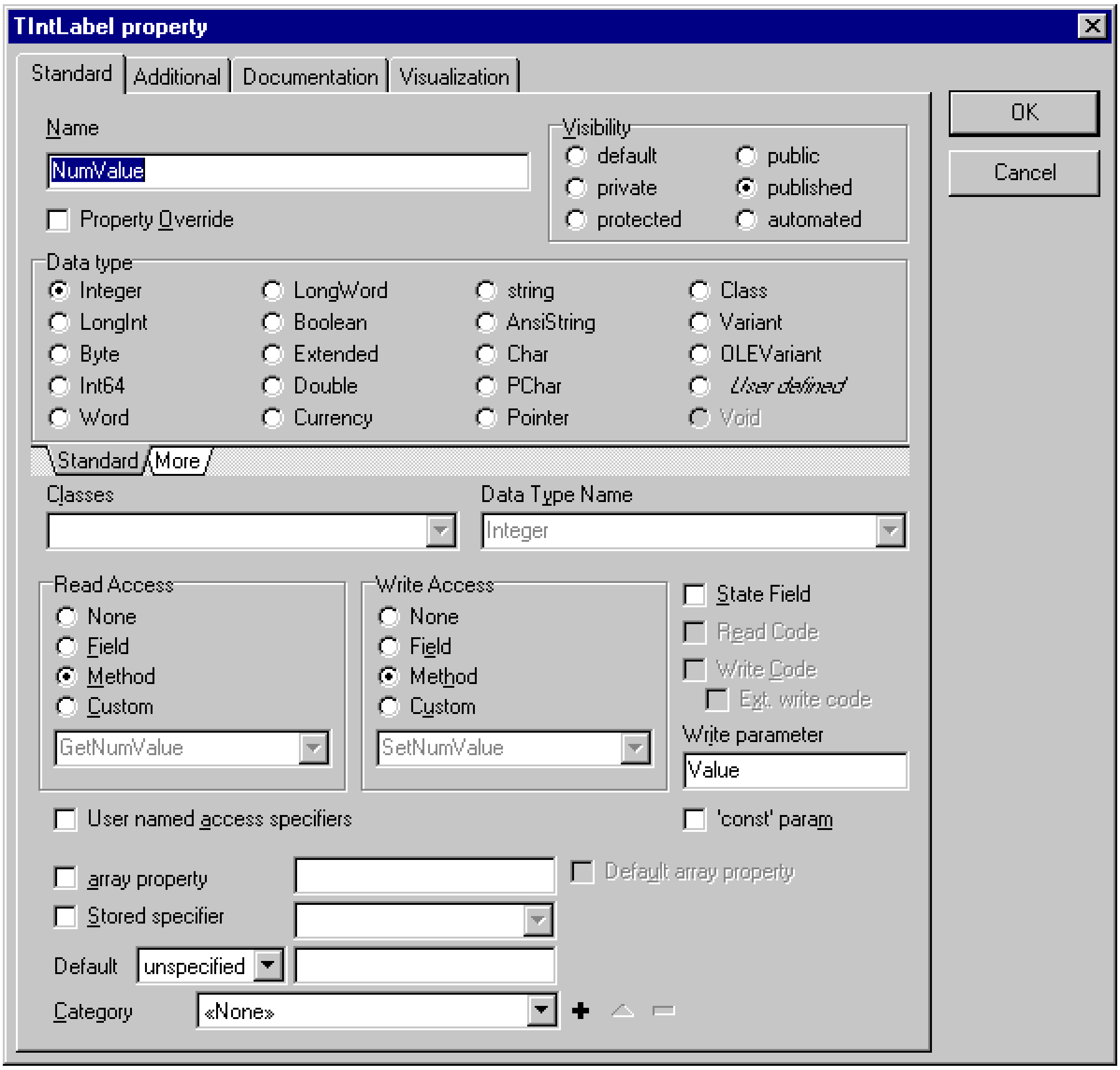
To do this we'll use the Class Members view - the bottom left window in the main window. Class Members are the fields, methods, properties (and event type properties) that make up a class's interface. The Class Members view is depicted here.



«All categories»

Pt St GetNumValue: Integer;
Pt St SetNumValue(Value: Integer)
Pu Ov Create(aOwner: TComponent
Pb RW NumValue : Integer

1. In this view all members for the currently selected class are displayed. Filters on *type* (Field, Method etc.), *visibility* (private, protected etc.) and *category* let you filter which members are displayed. Reset the filters by selecting ‘Reset Filter’ from the pop-up menu or by clicking the buttons ‘Show all types’ and ‘Show all visibilities’. All filter buttons should be in a ‘down’ state now, except of course the ‘Show all..’ buttons which do nothing but (p)reset the filters. Make sure the category filter shows <all categories>. The member list is still empty because we didn’t create any new Class Members yet. Note that the filter layout can be toggled using the popup menu ‘filter layout or double clicking on the filter area.
2. Click the ‘Add property’ button.
3. The property editor dialog will appear. See picture below.
4. Enter `NumValue` as the property’s name.
5. Select the visibility ‘published’.
6. Make sure the property’s data type is ‘Integer’.
7. Select for Read Access ‘Method’. This defines that the property has read access and you want to use a method to access it, rather than a field.
8. Select for Write Access ‘Method’ This defines that the property has write access and you want to use a method to access it, rather than a field.
9. Leave the other settings in their default values and click OK. In the property editor’s picture below the correct settings are displayed.



Name: NumValue

□ Property Override

Visibility
○ default ○ public
○ private ● published
○ protected ○ automated

Data type
● Integer ○ LongWord ○ string ○ Class
○ LongInt ○ Boolean ○ AnsiString ○ Variant
○ Byte ○ Extended ○ Char ○ OLEVariant
○ Int64 ○ Double ○ PChar ○ *User defined*
○ Word ○ Currency ○ Pointer ○ Void

Classes:

Data Type Name: Integer

Read Access
○ None
○ Field
● Method
○ Custom
GetNumValue

Write Access
○ None
○ Field
● Method
○ Custom
SetNumValue

□ State Field
□ Read Code
□ Write Code
□ Ext. write code

Write parameter: Value

□ User named access specifiers

□ 'const' param

□ array property

□ Default array property

□ Stored specifier

Default: unspecified

Category: «None»

Now have a look again in your Class Members view:
You’ll not only see a property `NumValue`, but also two property access methods `GetNumValue` and `SetNumValue`. This is because properties create and update their access fields and methods automatically. Now that saves time!

The `TIntLabel` class’s interface is now defined, but the methods `GetNumValue` and `SetNumValue` still need to be implemented.

Implementing methods

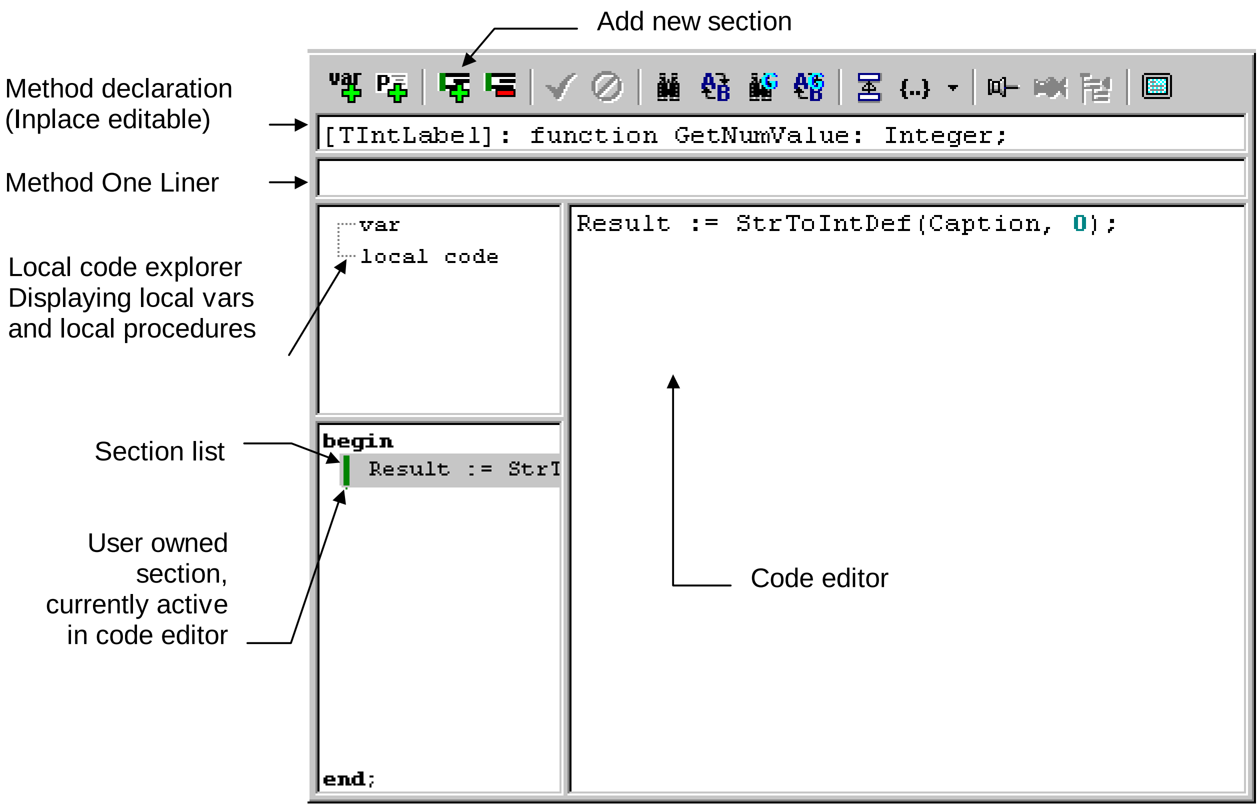
In our example we will need to add code to the implementation of the methods `GetNumValue` and `SetNumValue`. This code should be something like:

```
function TIntLabel.GetNumValue: Integer;
begin
```

```
  Result := StrToIntDef(Caption, 0);
end;

procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value);
end;
```

To add code to a method’s implementation you use the (Method) Implementation view.

1. Select the method you want to implement in the Class Member view, in this case the method `GetNumValue`.
2. To make the Method Implementation view visible, select ‘View|Implementation’.



```
[TIntLabel]: function GetNumValue: Integer;
```

```
Result := StrToIntDef(Caption, 0);
```

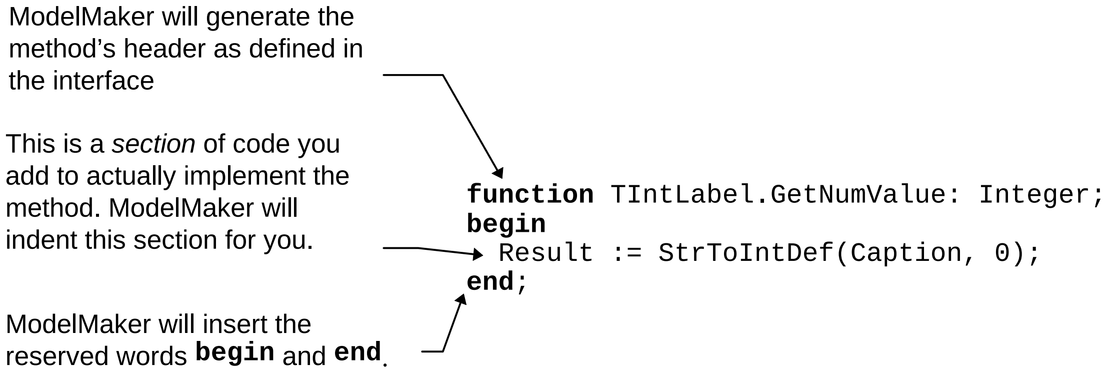
```
var
local code
```

```
begin
  Result := StrT
end;
```

Add new section
Method declaration (Inplace editable)
Method One Liner
Local code explorer Displaying local vars and local procedures
Section list
User owned section, currently active in code editor
Code editor

The picture above shows the Method Implementation view. This editor is perhaps the element of ModelMaker that is the most different from other editors. To understand how this editor works you need to know a little more about how ModelMaker generates code for a method’s implementation.

Let’s have a closer look at the `GetNumValue` method. This is just a simple method, not containing any local variables or local procedures.



ModelMaker will generate the method’s header as defined in the interface

This is a *section* of code you add to actually implement the method. ModelMaker will indent this section for you.

ModelMaker will insert the reserved words **begin** and **end**.

```
function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0);
end;
```

The body of a method’s implementation consists of a list of local variables, local procedures and *sections* of code which implement the block between **`begin..end`**. A section can take up any number of lines of code. All sections together make up the actual implementation. Using sections, ModelMaker is able to identify certain lines of code within the body. This is for example used to automatically add and update a call to the inherited method, as we’ll see later.

On the left side of the method code editor the complete method’s code is displayed, although maybe *collapsed* if necessary. On the right we find the actual code editor. It is used to edit the section of code selected in the sections list. The same editor is also used to edit the local procedures code.

As we see in the above picture, the only thing we need to do, is add a section of code containing the statement:

```
Result := StrToIntDef(Caption, 0);
```

To do this, first create a new section. If you didn’t change the code options settings in ‘Options|Code options’ a new section is automatically created if the method does not contain any sections yet.

1. If necessary, add a new section by clicking the ‘Add section’ button.
2. Enter the statement in the code editor. There’s no need to indent the code with spaces since this will be done automatically by ModelMaker.
3. Click the ‘save code’ button. This is not really necessary, since ModelMaker will automatically save the section as soon as you select a new section or a new method.

Notice that the section is now also displayed in the section list, and is marked with a green line. This green line informs you that *you* created this section and are it’s *owner*. Red lines indicate that a section is not owned by you, but, for example, is inserted by a pattern which has the only rights to update it. If a section contains more lines than the current ‘Fold height’ (adjustable in the Environment options tab Editors), the section will be collapsed. Collapsed sections are marked with a second purple line with a mark on the collapsing position. More about this later.

Now you should be able to implement the `SetNumValue` method the same way: select the method in the Class Members view, add a section if necessary and enter your code.

Although we have finished implementing the class `TIntLabel` for now, all code exists only in the ModelMaker model, so the next thing to do is to generate a source file.

Creating a source file

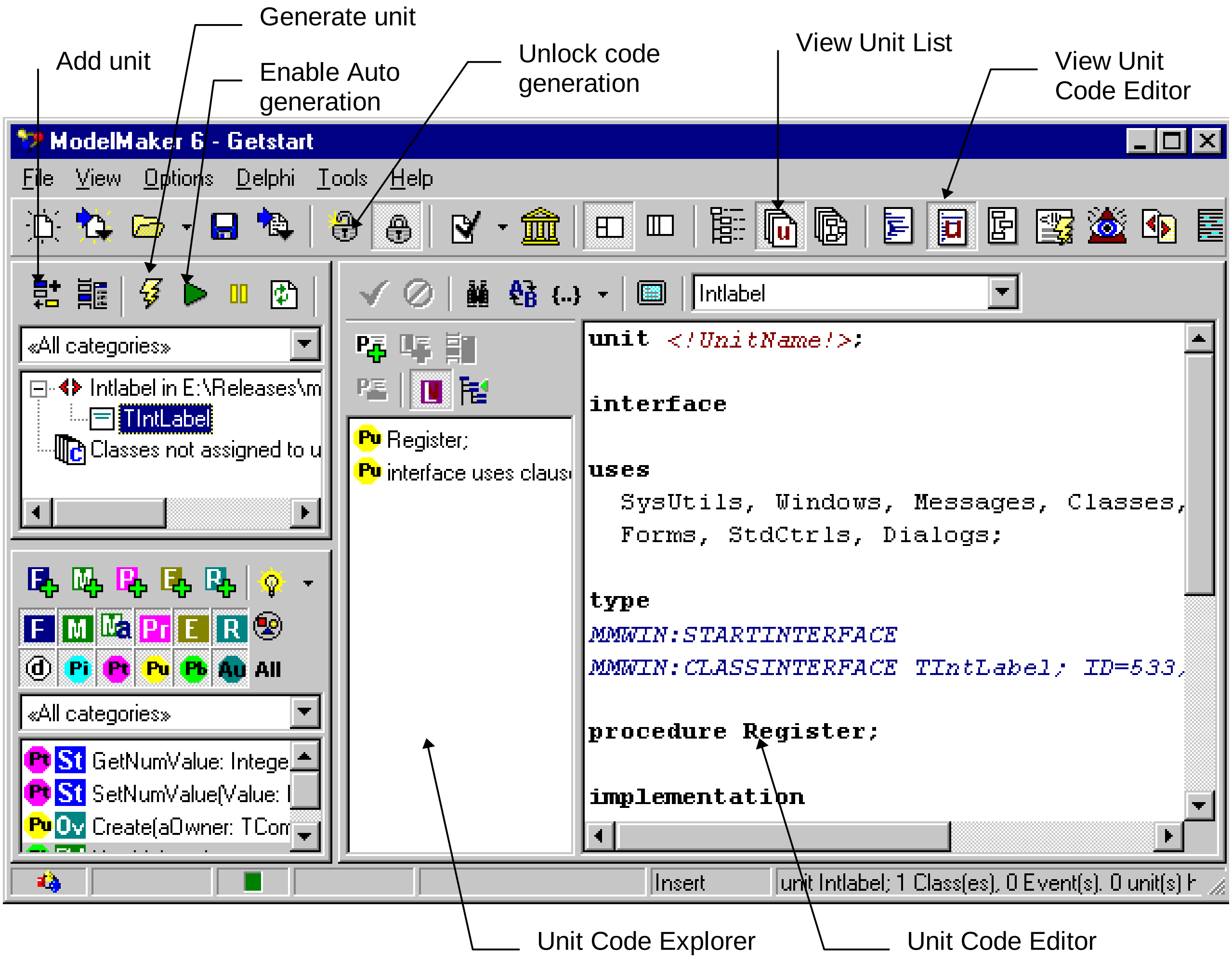
Units are the gateways to source files on disk. They provide a link between all data in a ModelMaker project such as classes, method implementations etc. and an Object Pascal style unit file which Delphi is able to compile.

Creating a Unit

In this example we need to create a unit which, after it has been generated, should look something like:

```
unit IntLabel;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue: Integer read GetNumValue write SetNumValue;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents(’MM Demo’, [TIntLabel]);
end;

function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0);
end;

procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value);
end;

end.
```

To create a new unit we use the Unit List view which is depicted below,



Pt St GetNumValue: Intege
Pt St SetNumValue(Value: I
Pu Ov Create(aOwner: TCom

```
unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes,
  Forms, StdCtrls, Dialogs;

type
MMWIN:STARTINTERFACE
MMWIN:CLASSINTERFACE TIntLabel; ID=533;

procedure Register;

implementation
```

unit Intlabel; 1 Class(es), 0 Event(s), 0 unit(s) I

1. To make the Unit list view visible, select ‘View|Units’. Repeat this for the Unit Code editor.
2. In the Unit list click the ‘Add unit’ button, a unit properties dialog will now appear.

In the Unit editor dialog you define:

1. Leave the source path alias `<no alias>` unchanged.
2. The source file name (the full path including drive and folders), to define the path you could use an source alias, but for now just click the browse button and locate the `\ModelMaker\6.0\TEST` folder (or `ModelMaker\6.0\Test` depending on the base path you installed ModelMaker in) and enter the file name `IntLabel.PAS`.
3. On the tab sheet ‘Classes’ add the class `TIntLabel` to the list on the right either by dragging or by selecting it and clicking the ‘Add selected’(▸) button or by double-clicking.
4. Change the ‘VCL page’ from `<unregistered>` to ‘`MM Demo`’ by entering this name in the string grid.
5. Click OK.

We’ll see the newly created unit now listed in the unit list on the left. In this editor unit’s code containing a text which should look something like:

```
unit <!UnitName!>;
```

```
interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
MMWIN:STARTINTERFACE
MMWIN:CLASSINTERFACE TIntLabel; ID=7;

procedure Register;

implementation

procedure Register;
begin
MMWIN:CLASSREGISTRATION TIntLabel; ID=7; Page='MM Demo';
end;

MMWIN:STARTIMPLEMENTATION
MMWIN:CLASSIMPLEMENTATION TIntLabel; ID=7;

end.
```

For now, it is enough to understand that ModelMaker uses tags (like `MMWIN:CLASSINTERFACE`) to insert the interface, VCL registration and implementation of a class in otherwise ‘plain’ text that you define. There is one problem however: if you look at the **`uses`** clause in the interface, you’ll see that the unit `StdCtrls` which defines the ancestor class `TLabel`, is missing. That is because the default unit template we are using does not contain this unit. For changing this template refer to ‘Customizing ModelMaker’. For now, we will have to add `StdCtrls` manually.

To do this,

1. In the unit code editor add `StdCtrls` to the uses clause.
2. Click the ‘Save code’ button in the toolbar above the editor.

Generating the source file

To generate the source file and create or update a file on disk,

1. Make sure that code generation is not locked. Locking is explained later, for now make sure the button ‘unlock code generation’ in the ModelMaker toolbar is pressed down.
2. Click the ‘Generate current unit’ button in the unit list view.
3. Start Delphi (if it was not running already).
4. Either manually switch to Delphi or - much more instructive - click the ‘Locate in Delphi’ button in the main tool bar - or simply press Ctrl+F11. This will open the unit and locate the entity currently selected in ModelMaker.

The generated source file should look pretty much the same as we wanted it to be. (Differences may occur if you modified your file `DEFUNIT.PAS` in ModelMaker’s `\BIN` folder.)

We’re ready to debug the `TIntLabel` now, and install it in our VCL.
Before doing this it’s a good idea to save our model. This is very much like in other windows applications, so it won’t be explained here. We could use the `[installdir]\TEST` folder to

save this project. Practice shows that it is convenient to name your model after the main source file you create with it, or the main set of components. In this case `INTLABEL.mpb` seems an obvious name.

Adding the component to the VCL

Debugging your component

A good practice, is to debug your new component before adding it to the VCL. Do this for example by adding the source file to the current Delphi project (e.g. by using Delphi's project manager) and re-compile the project. This should at least filter out all syntax errors. Either use 'Compile' or 'Syntax Check' since Delphi does not always correctly manage the file's date/time and modified status if you just 'Run' the project.

Compiling errors

If you didn't make any mistakes, the unit should compile all right. If it doesn't: change the code in the appropriate place in ModelMaker. That is:

- For missing units in the unit's uses clause: the unit code editor.
- For any code not part of the class: the unit editor.
- For errors in the class's name or ancestor name: the Class view.
- For errors in the class's interface declaration: the Class Members view.
- For errors in a method's implementation: the Method Implementation view.

Switch to ModelMaker and fix the code. Use the integration expert's menu 'Jump to ModelMaker' to jump straight from Delphi's code editor to the corresponding position in ModelMaker. Finally, in the Unit list view click the button 'Generate' again and re-compile the Delphi project. If you are having trouble with this: look ahead where we are adding new behavior and editing code is explained.

Adding the component to the VCL

After debugging your new file, add it to the Delphi's VCL.
In *Delphi 1.0*: select menu 'Options|Install Components', select 'Add' and browse to find the unit `INTLABEL.PAS` in folder ..`\ModelMaker\6.0\TEST\`.
In *Delphi 2.0*: Select menu 'Component|Install', select 'Add' and browse to find the unit `INTLABEL.PAS`. Refer to your Delphi User guide for more information about installing components.
In *Delphi 3 and higher*: you must install the new component in a package. Select a new package called MMtest in the ..ModelMaker\6.0\Test folder. Please refer to your user guide for installing packages.

After recompiling the VCL the new TIntLabel component should be on the palette page where you registered it: *MM Demo*. To test it, add a TIntLabel component to a (new) form.

You can use the Object Inspector now to set the 'NumValue' property and see the caption change. But the component can be improved!

Improving the component in ModelMaker

If you watch carefully, you'll see that a TIntLabel when dropped on a form, initially has the caption 'IntLabel1' rather than '0'. The NumValue however is 0. This is conflicting and not very nice. To improve the component we'll have to override the constructor Create like this: (refer to Delphi's on-line help for the TControl.ControlStyle property)

```
constructor TIntLabel.Create(AOwner: TComponent);
begin
 inherited Create(AOwner);
  { Don't let the Object Inspector set a caption }
  ControlStyle := ControlStyle - [csSetCaption];
  { Instead pre-set the Caption ourselves }
  NumValue := 0;
end;
```

To do this we need to return to ModelMaker.

Keep editing your code in ModelMaker

We could of course change TIntLabel's code in Delphi, but then the ModelMaker model and the modified source file would be out of sync. The next time we would (re-)generate the file from within ModelMaker, the changes made in the Delphi Editor will be lost. Of course, if we do not intend to maintain our code any longer in ModelMaker that's fine, but we won't benefit the advantages ModelMaker offers during maintenance and documentation. And although it may seem a burden at first, after getting used to it, the benefits of keeping the master code in ModelMaker are much higher than the costs. So resist the itch in your fingers and return to ModelMaker now.

Overriding the constructor Create

To override the constructor Create, we could add a method in the Class Members view clicking 'Add method', name it 'Create' adjust it's other attributes such as parameters 'AOwner: TComponent', method kind 'constructor', etc. but overriding methods can be done far more easy. The only thing is: in order to override a method (or property) the method to be overridden must exist in the model. If you used the default project template as was shipped with ModelMaker (which also contains the class TComponent), the virtual constructor TComponent.Create is in your model with the correct attributes, ready to be overridden.

To do so:

1. In the Members view tool bar click the 'Wizards' button, and

2. Select ‘Method override wizard’. Alternatively use the same function from the Wizards popup sub menu.
3. In the ‘Override Methods’ dialog select the ‘`Create`’ method.
4. Make sure the option ‘Call inherited method’ is checked. This will instruct ModelMaker to add a section of code containing a call to the inherited method.
5. Click OK.

In the Class Members view we’ll see that a method called `Create` is added to the list. Just for your information you may check the methods attributes by selecting it in the member list and clicking the ‘Edit Member’ button, or double clicking it in the member list.

Notice that all relevant attributes are copied from `TComponent.Create`:

- The methods name is `Create`.
- The parameter list is `AOwner: TComponent`.
- The method is ‘public’.
- The data type is ‘void’.
- The method type is ‘constructor’.
- The binding kind is ‘override’ (since `TComponent.Create` is virtual).
- The option ‘Call inherited’ is checked because we checked the option ‘Call inherited method’ running the override wizard.
- The option ‘Inheritance restricted’ is checked. If this option is checked, the method will automatically be updated to reflect any changes applied to the overridden method in the ancestor class.

Click Cancel to leave the method in it’s original state.

Implementing Create, non-user sections in code

To implement the method `Create` switch to the Method Implementation view again. Notice that in the section list on the left, already one section is added containing the code:

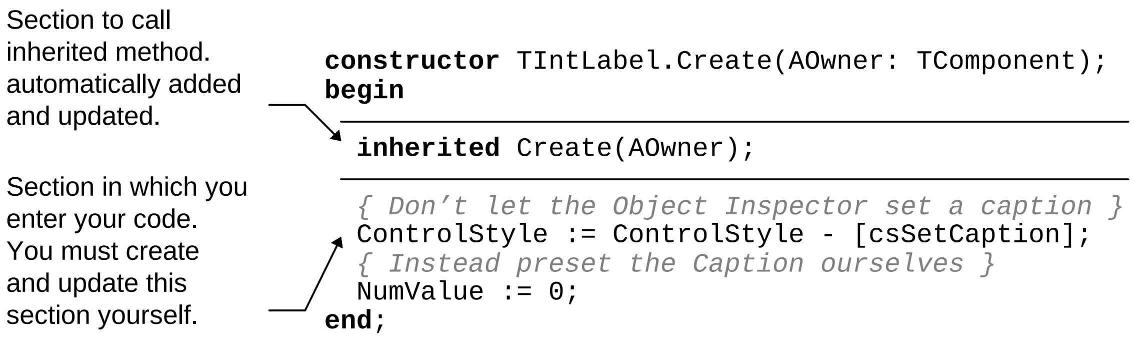
```
  inherited Create(AOwner);
```

This section is marked with a red line, indicating that we cannot edit it’s contents. The section was added because the method’s option ‘Call inherited’ is checked.

To add the other lines of code,

1. Add a new section by clicking the ‘Add section’ button.
2. Enter the code in the code editor on the right.
3. Click the ‘Save code’ button

In the section list on the left we'll see the complete implementation of `Create`.



Section to call inherited method. automatically added and updated.

Section in which you enter your code. You must create and update this section yourself.

```
constructor TIntLabel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Don't let the Object Inspector set a caption }
  ControlStyle := ControlStyle - [csSetCaption];
  { Instead preset the Caption ourselves }
  NumValue := 0;
end;
```

Instant code generation

If we have a look in the Delphi editor, we'll see that the source file has not been updated yet. To do this we need to regenerate the unit. Therefore switch to the Unit list view again.

Regenerating the unit could be done by clicking the 'Generate' button again, but it is more instructive to demonstrate ModelMaker 's *instant code generation* feature. Rather than having to manually regenerate a source file whenever something has changed, it is possible to '*Enable Auto generation*' for a unit. The source file will then be regenerated each time anything changes in the Model that affects the source file. It is a nice feature that ModelMaker not only regenerates the source file, but also instructs Delphi to reload the file if it's opened in Delphi's code editor. Refer to Integration with Delphi.

To watch this:

1. Make sure you have the `INTLABEL.PAS` file loaded and is on top in the Delphi code editor.
2. Click the 'Enable auto generation' unit button in the Unit view tool bar.
3. See how the Delphi editor now reflects the last changes in your file.

Now return to ModelMaker again, and let's play around:

1. Switch to the Class Members view.
2. Edit the Create Method (Double click or click the Edit button).
3. Now uncheck the 'Call inherited' option and click OK.
4. Watch the code being updated in Delphi.
5. Now check the 'Call inherited' option again.

To have a closer look at the Method Code editor,

1. In the Create method's section list you can drag sections up and down, do this and see how Delphi's code editor follows your changes.

Lets have play around with the units view:

1. Make sure the units view is in ‘display as tree mode’ (popup menu)
2. Select the class `TIntLabel` by clicking it.
3. Press the Del key once, this will remove the class from the unit. The class is still in the code model. In fact, the class is listed under the ‘classes not assigned to units’ node. Check the code in the IDE: you’ll see that the entire interface and declaration have been removed.
4. Drag the class on the IntLabel unit. This will add the class to the unit again. Note that the VCL Component registration page is now reset to ‘none’. Edit the unit (double click unit or use toolbar) to change this back to ‘MM Demo’.

After you have played around with the instant code generation feature, make sure the `TIntLabel` class still is as you want it to be. In order to actually see the improved behavior,

1. Rebuild your VCL in Delphi.
   Delphi 1: menu ‘Options|Rebuild library’.
   Delphi 2: menu ‘Component|Rebuild library’.
   Delphi 3 and higher: recompile the package MMtest.dpk.
2. Remove any old `TIntLabel` components from forms.
3. Add a new `TIntLabel` to a form and notice how the Caption is set to ‘0’ now.

Documenting your component

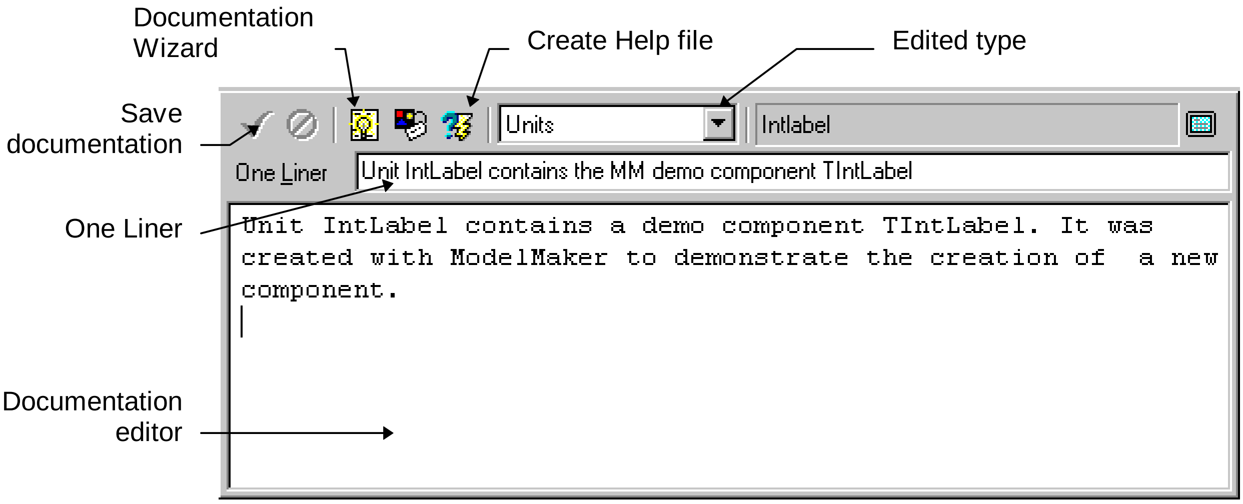
ModelMaker not only supports source code generation for your component, but it has also advanced wizards and generators to document your component. These include

- Documentation wizard which inserts basic standard documentation for all members in a class.
- In source documentation.
- Help file generation.
- Instant visualizing in class diagrams. Although creating diagrams is usually done in the design process, it is also possible to create diagrams from existing code.

To demonstrate these features we will now create a help file and a class diagram for the `TIntLabel` component.

Adding documentation to your component

Each unit, class, all members of a class, ebent types and symbols in diagrams can be documented with a short description named One Liner and a longer text named Documentation. For editing One Liner and documentation we’ll use the Documentation view. Alternatively we could have used the floating documentation window which is available from the main menu “Views”.



Unit IntLabel contains the MM demo component TIntLabel

Unit IntLabel contains a demo component TIntLabel. It was created with ModelMaker to demonstrate the creation of a new component.

To make this view visible: Select menu ‘View|Documentation Editor.

With this Documentation editor you add a One Liner and a more descriptive text to each unit, class and member (method, property etc.). That can be quite a job, so ModelMaker includes a documentation wizard, which does some of the nasty work for you. This wizard will insert pieces of documentation in the currently selected class.

To demonstrate this:

1. Select the class `TIntLabel` in the Classes or Units view.
2. In the Documentation view click the button ‘Documentation Wizard’.
3. Click OK to confirm creation of standard documentation.
4. In the drop-down box ‘Edited type’ select ‘class members’.
5. In the Class Members view select the `GetNumValue` method.

What you see now in the documentation editor is that the wizard inserted text like:
GetNumValue is the read access method for the NumValue property.
Usually this is sufficient to document `GetNumValue`, since you will be documenting the exact meaning of the property `NumValue` and this avoids redundancy.

Selecting the method `SetNumValue` in the Class Members view makes the documentation for `SetNumValue` visible:
SetNumValue is the write access method of the NumValue property.
Again this is usually sufficient to document the `SetNumValue` method.

What remains to be done is documenting the constructor `Create` and the property `NumValue`. But here too, the wizard inserted already some useful text.

Select the documentation for the constructor `Create` and change this to:
Constructor Create overrides the inherited Create. First inherited Create is called, then the Caption is pre-set to 0, reflecting the initial NumValue state. ControlStyle is modified to exclude csSetCaption.

Now select the `NumValue` property’s documentation and change this to:
Property NumValue is read/write at run time and design time.

It reads and writes the Caption property as an Integer.

To edit the documentation for the class `TIntLabel`:

1. Make sure class `TIntLabel` is selected in the Classes or Units view,
2. In the Documentation view, select ‘classes’ in the drop-down box ‘Edited type’.

Enter the text:
TIntLabel is a simple TLabel descendant created with ModelMaker.
It adds the property NumValue which reads and writes the Caption property
as an Integer.

To edit the documentation for unit IntLabel which contains the `TIntLabel` class:

1. Make sure the unit `IntLabel` is selected in the Units view.
2. In the Documentation view, select ‘units’ in the drop-down box ‘Edited type’.

Enter the text.
Unit IntLabel contains a demo component TIntLabel. It was created with
ModelMaker to demonstrate the creation of a new component.

The unit `IntLabel` is now completely documented. Documentation is typically used to create a helpfile or for in-source documentation. Third party plug-in experts use the ModelMaker ToolsApi to output documentation to other formats.

You add One Liners (short, single line descriptions) the same way. In the Views menu you’ll find a ‘Floating documentation’ view. This view can be used to insert One Liners and documentation too. This view can be docked or stay floating. The edited entity type in this view is automatically updated to reflect the last focused view in ModelMaker.

Creating a help file

ModelMaker can create a Borland style help file from your documentation. This includes the generation of the Borland `/B` keywords which are necessary for interaction with Delphi’s on-line context sensitive Help. Help files are generated from unit’s. In our example we’ll create a help file for unit `INTLABEL`.

ModelMaker let’s you select the visibilities you want to include in your help file. These are:

- ‘User’ (public, published, automated and default)
- ‘Component writer’ (user visibilities plus protected)
- ‘Developer’ (all visibilities)

The default visibility ‘User’ is the most restricted, since this will include only help for the public, published or automated interface of a class. Use this filter to create a help file you distribute with your components. Selecting the ‘Component writer’ visibilities will also include help for the protected interface. This is the type of help file you would typically distribute with components if other developers should be able to derive a descendant class from your component. The last selection includes also the private details (typically fields and or property access methods etc.) which you might want to have documented internally.

To create a help file for the `TIntLabel` component,

1. Make sure the unit `IntLabel` is selected in the Units view.
2. In the Documentation view, click the button ‘Create help file’.
3. In the ‘Create help file’ dialog you’ll be prompted to enter a file name for the unit’s RTF file. A help project file with the same name and extension `.HPJ` will automatically be created. In our example enter `[installdir]\TEST\INTLABEL.RTF.`
4. In the same dialog, select the visibilities you want to include in your help file. In this example we’ll go for the ‘Component writers’ visibility.
5. Leave the reformat paragraphs option checked and Click OK.
6. Open the explorer (file manager) and notice that both `INTLABEL.RTF` and `INTLABEL.HPJ` have been created.
7. Run your Delphi help compiler with the `INTLABEL.HPJ` project. Be aware that you need to use the help compiler that was shipped with the Delphi version you want to create help for. For *Delphi 1.0*: Use `DELPHI\BIN\HC31.EXE` in a DOS box and run it from the folder your HPJ file is in. For example: `DELPHI\BIN\HC31.EXE INTLABEL.HPJ`
For *Delphi 2, 3 and higher*: Use the `Delphi 2(or 3 / 4/5/6).0\HELP\TOOLS\HCW.EXE` to compile your project: double clicking the `INTLABEL.HPJ` file in the explorer should be enough.

The help files have been created now. You might want to have a look at them, using windows help. Double clicking the newly generated `INTLABEL.HLP` starts help.

The Help File Generator source is available as plug-in expert on request. It can be extended to create help for multiple units at once etc.

Integrating your help files with Delphi’s on line help

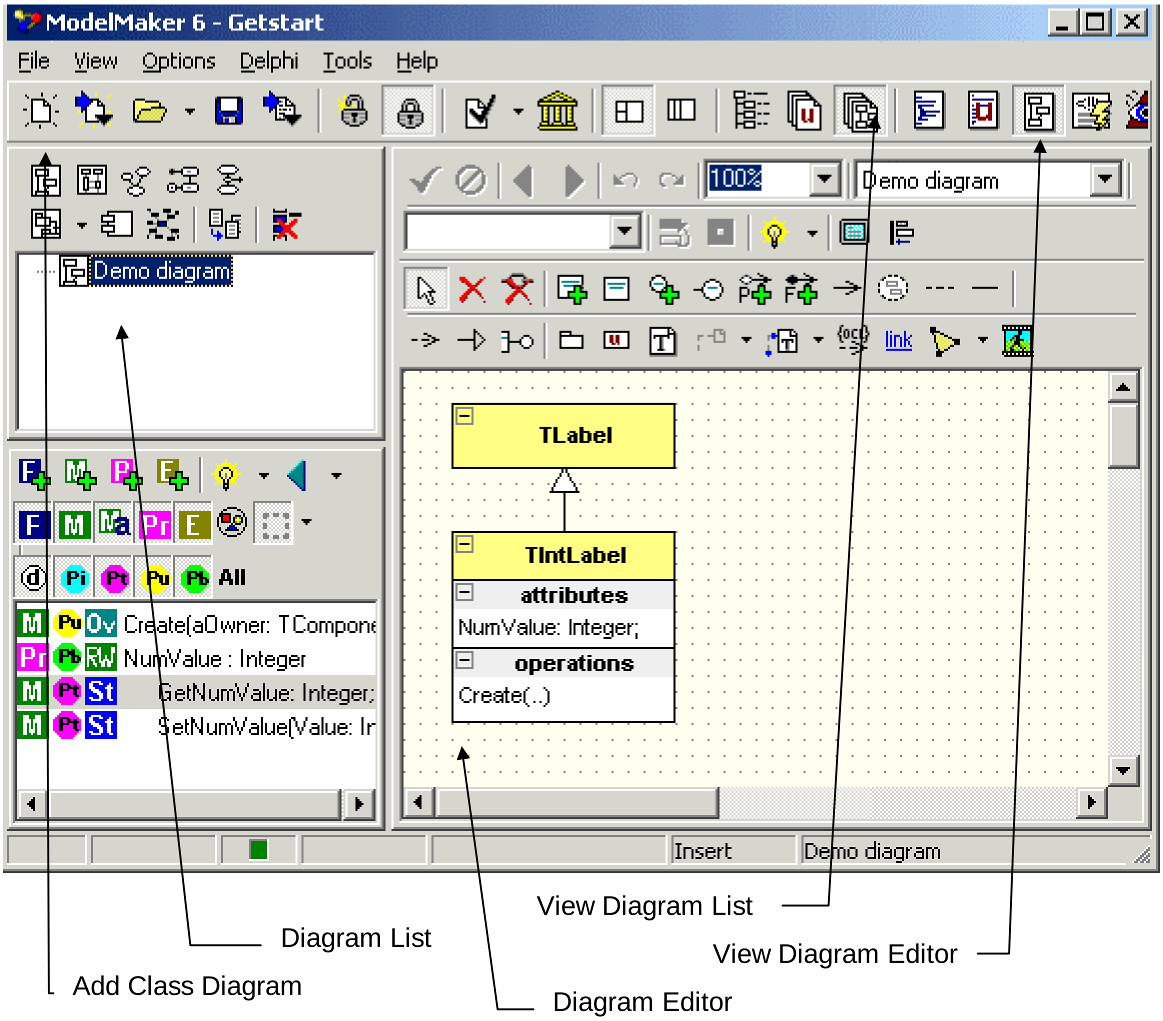
To be able to invoke Delphi’s help on your `TIntLabel` component you must integrate your `INTLABEL.HLP` help file with the Delphi 1 and 2 on line help. To do this:

1. Generate key words using `KWGEN.EXE`
2. Install help using `HELPINST.EXE`

Installing help in Delphi 3 and higher is documented in Delphi’s Component Writers Guide / User’s Guide.

Documenting the design in a diagram

Although creating diagrams is usually done in the first stages of the design process, it is also possible to create diagrams from existing models using ModelMaker’s instant visualization feature. A list of diagrams in the model is edited in the Diagram List view. The actual diagrams are edited in the Diagram Editor.



Make the Diagrams List view and Diagram Editor visible:

1. Select menu 'View|Diagrams'. This will make the diagram list visible in the top left window.
2. Select menu 'View|Diagram Editor'. This will make the diagram editor visible in the editor pane on the right
3. In the Diagram list create a new class diagram by clicking the 'Add Class diagram' button.
4. You can inplace edit the diagram's name. Enter 'Demo diagram'. The name is intended only to distinguish different diagrams.

In the newly created diagram we'll demonstrate ModelMaker's instant visualization feature:

1. Make sure the Classes or Units view is visible (View|Classes or Units)
2. Drag the class `TLabel` from the Classes or Units view and drop it on the diagram editor.
3. Do the same with `TIntLabel`.
4. Save the Class diagram by clicking the 'Save diagram' button.

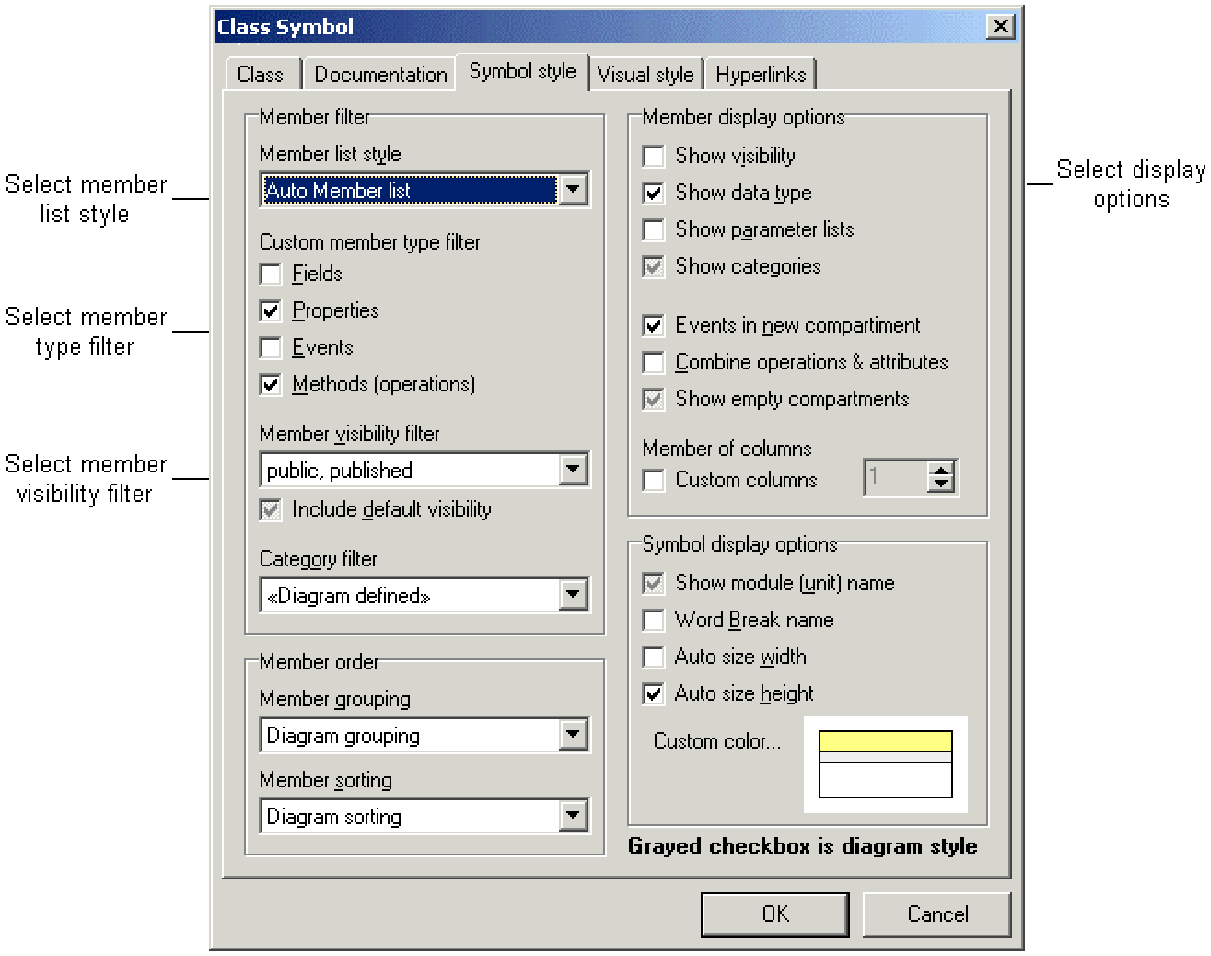
Notice how the inheritance relation `TIntlabel = class (TLabel)` is automatically visualized. If appropriate, ModelMaker will also visualize 'uses' relations if a class is dragged onto a diagram.

You can copy the current selected diagram to the clipboard (in WMF format) and paste it in your word processor to document your design. To do this: press `Shift+Ctrl+C` or use the local menu ‘Export as Image’|‘Clipboard’. Alternatively you could export the image to a file (bmp, wmf and jpg). To print the diagram, press `Ctrl+P` or use the pop-up menu.

Symbol styles in Diagrams: displaying members

ModelMaker supports many UML styles of displaying classes and interfaces: show module (unit) names, show members, collapse interfaces etc. To demonstrate a few and give you an idea of what is possible we’ll change the symbol style for the `TIntLabel` class symbol. Symbol styles define how a symbol is displayed in a diagram. Note: the visual appearance (colors, fonts etc) is controlled by the visual style which is not part of the symbol style; check this manual for details. Default all symbol styles are “as defined in current diagram”. The diagram symbol style defaults to “as defined in the current project”. This gives you the possibility to change style on any level you like: just a single class symbol, all class symbols in a diagram or all diagrams in a project.

Here we’ll change the symbol for TIntLabel only. Double click on the TIntLabel class symbol. This will show the class symbol editor.



Select member list style

Select member type filter

Select member visibility filter

Select display options

Class Symbol

Member filter

Member list style: Auto Member list

Custom member type filter
- ☐ Fields
- ☑ Properties
- ☐ Events
- ☑ Methods (operations)

Member visibility filter: public, published
- ☑ Include default visibility

Category filter: «Diagram defined»

Member order

Member grouping: Diagram grouping

Member sorting: Diagram sorting

Member display options
- ☐ Show visibility
- ☑ Show data type
- ☐ Show parameter lists
- ☑ Show categories
- ☑ Events in new compartment
- ☐ Combine operations & attributes
- ☑ Show empty compartments

Member of columns
- ☐ Custom columns 1

Symbol display options
- ☑ Show module (unit) name
- ☐ Word Break name
- ☐ Auto size width
- ☑ Auto size height

Custom color...

Grayed checkbox is diagram style

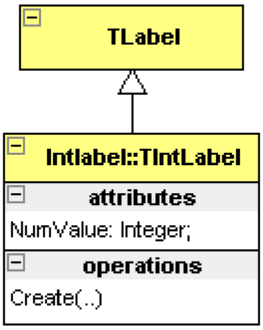
First you define ***which*** members are displayed. In this dialog change the Member list style from “Diagram Auto Member list” to “Auto Member list”. This style will automatically add all members defined by the member filter in this editor. In the Member filter check

“Properties” and “Methods”. Change the visibility filter to public and published members only. This will suppress display of the protected property access methods GetNumValue and SetNumValue.

Then you define ***how*** members are displayed. Use the Member display options to modify this. Check “Show Data type” and “Events in new Compartment”. Note that a grayed option reverts the option to the parent (diagram) style.

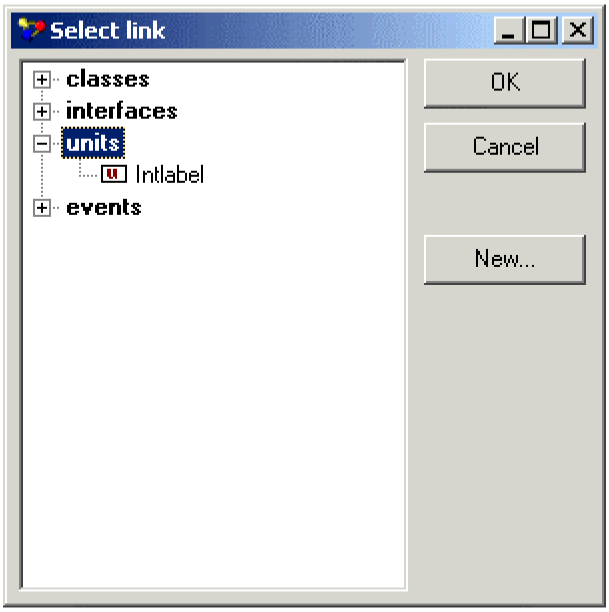
To display the name of the unit that contains this class in the symbol name compartment, check the “Show module name” option.

After clicking OK you should have a diagram that looks something like this (you can stretch a class symbol as needed depending on the Auto Size options – check the class symbol dialog). The property NumValue is displayed in the ‘attributes’ compartment and the method Create is displayed in the ‘operations’ compartment. Likewise events can be displayed in a separate compartment or combined with the attributes. If you wish you could even combine all members into a single compartment.

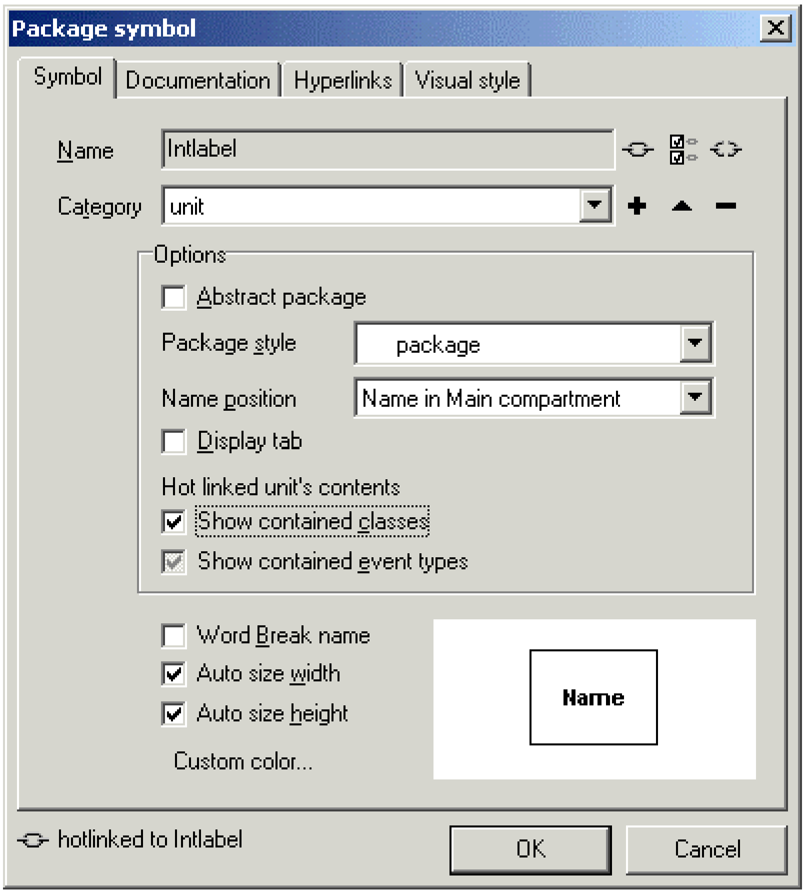


Visualizing the unit IntLabel.pas

Just like classes can be visualized by class symbols, the unit IntLabel which contains TIntLabel can be visualized. To do this, click the “Add Unit Package” tool on the diagram editor tool bar and then click on the diagram. The following dialog will appear which lets you select the unit to visualize in the package. In this dialog, select “IntLabel” and click OK.



Now the Package Symbol editor will be visible, which is used to edit the visualization of the linked unit.



Name: Intlabel

Category: unit

□ Abstract package

Package style: package

Name position: Name in Main compartment

□ Display tab

☑ Show contained classes

☑ Show contained event types

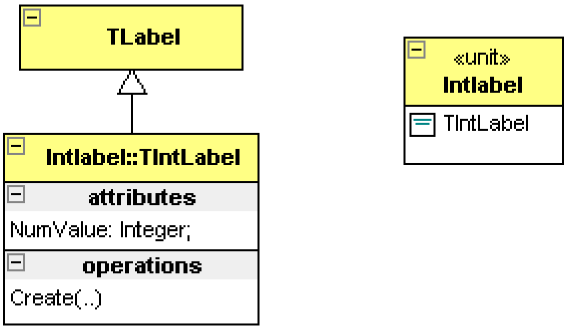
□ Word Break name

☑ Auto size width

☑ Auto size height

hotlinked to Intlabel

In this dialog, Make sure the “Show contained classes” option is checked (not grayed) and click OK. Your diagram should now look something like the picture below.

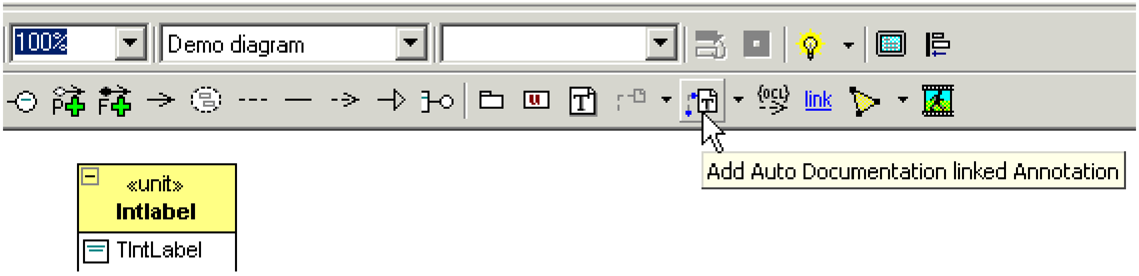


The package symbol displays it's contained classes (TIntLabel) and has a stereotype (category) named <<unit>>. Of course, if you add more classes to the unit, the symbol will be updated automatically.

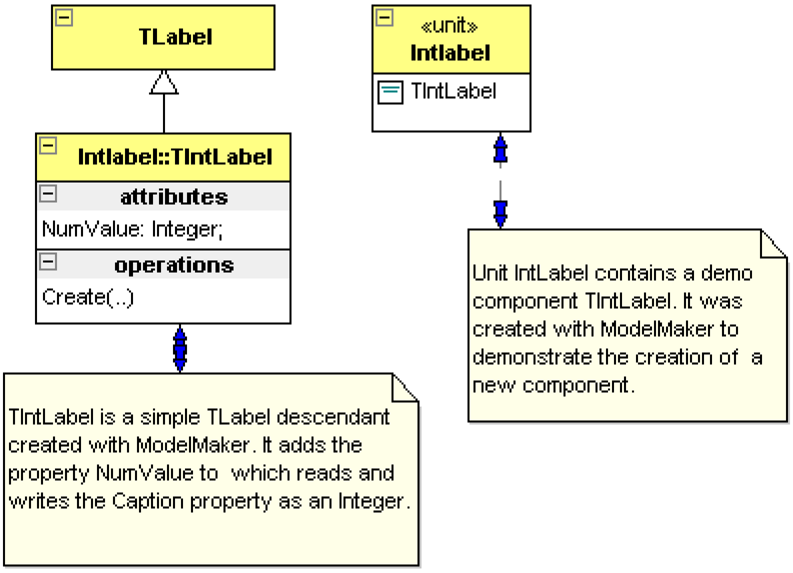
Visualizing the Documentation

The UML uses Annotation symbols to add notes to diagrams. ModelMaker supports hotlinking annotations to symbol documentation (or OneLiners). This is a two way hot link: the annotation text will automatically show the symbol's documentation and editing the annotation text will update the symbol's documentation. To demonstrate this, we'll add a linked annotation to the class symbol TIntLabel and to the unit package symbol IntLabel.pas.

On the diagram editor toolbar select the "Add Auto Documentation linked Annotation" tool. This tool allows three link styles which can be selected with the drop down button next to it. Links styles are: passive (not linked), documentation and one liner. Make sure the Documentation style is selected.



Then after selecting this tool, click on the class symbol and drag the mouse below it. This will create the link and annotation. Repeat this for the package symbol that is linked to unit IntLabel. Your diagram will now look something like this.


TLabel
«unit»
Intlabel
TIntLabel
Intlabel::TIntLabel
attributes
NumValue: Integer;
operations
Create(..)
Unit IntLabel contains a demo component TIntLabel. It was created with ModelMaker to demonstrate the creation of a new component.
TIntLabel is a simple TLabel descendant created with ModelMaker. It adds the property NumValue to which reads and writes the Caption property as an Integer.

Now try to edit the documentation for TIntLabel in the annotation and see it change in the class dialog and the diagram. To do this, click TIntLabel's annotation and press F2. This invokes the annotation's inplace editor. Change the text to your liking and after pressing the Enter key, check the documentation tab in the class editor dialog (classes view). Similar to linking the documentation, a symbol's One Liner can be linked to an annotation.

Summary

In this chapter you got a first glimpse of some basic features in ModelMaker.

- ModelMaker is all about creating classes.
- It is important to start with the right project template especially if you want to override methods or properties.
- The interface of a class consists of Class Members (fields, methods and properties).
- Properties create and update their access fields and methods.
- Method code consists of sections, which can either be created automatically or manually.
- A Unit provides a link with a source file. Units can have 'auto generation' enabled to instantly reflect any change in the model to the source file.
- ModelMaker's IDE integration experts will take care of reloading units in Delphi's code editor.
- To jump from ModelMaker to the IDE and back, use the Locate in Delphi and Locate in ModelMaker commands.
- It's easy to document a design using the documentation wizard.
- Help files can be created for a unit, which are ready to integrate with Delphi's on-line help.
- Instant visualization in class diagrams can be used to document your design.
- Class symbols can automatically display contained members, package symbols can automatically display classes contained by a unit (source module).

- Symbols can be hotlinked to annotations, which will then display documentation or One Liners.
- Diagrams support multiple symbol styles that can be defined at different levels.

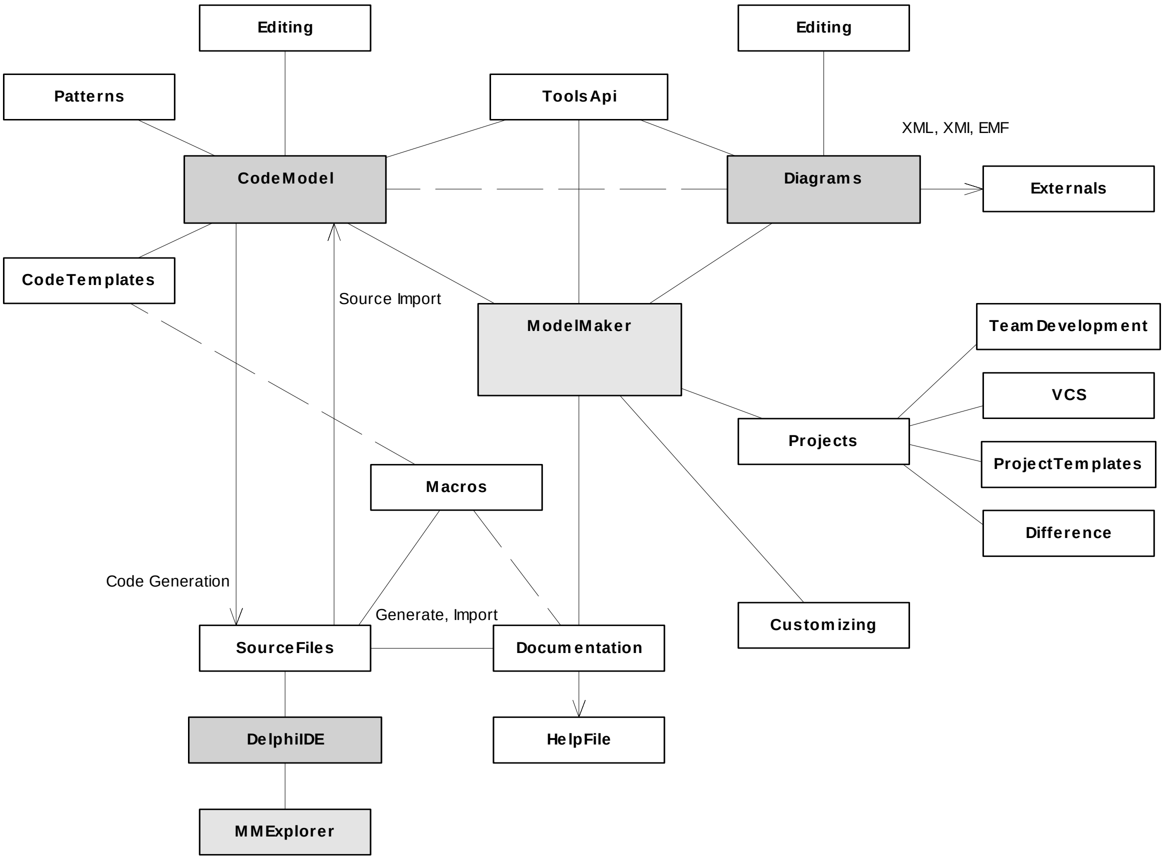
Where to now?

Now you’ve seen ModelMaker’s basic mechanisms. You can have a look at the following topics:

- There is another step-by-step example in the Design Patterns manual that demonstrates the use of design patterns. You should have enough background now to work through this demo.
- ModelMaker’s *Basic Concepts* page 36.
- The other chapters in this manual focus in greater detail on common aspects of ModelMaker such as diagrams, code generation and import.

Basic Concepts

Overview



This picture gives an impression of ModelMaker’s main parts and how they relate.

The two main entities are the Code Model and the Diagrams. Around them you’ll find importing and generating source code, interaction with the Delphi IDE, saving models etc. Going around this picture more or less anti-clockwise we’ll see:

The ***Code Model*** contains the classes, members, units etc. that map to the corresponding concepts in Delphi’s Object Pascal. The Classes view, Members view, Method Implementation view and Units (Code) view all deal with visualizing and manipulating the code model directly.

Code Generation is the process of creating an Object Pascal source file containing classes, members and unit code. ***Units*** provide the link between the Code Model and a source file. A unit contains classes and/or event type declarations plus user defined ***unit code*** such as module procedures. Refer to Code Generation, page 45, for a more detailed description of source code generation. Code generation is typically, but not only, controlled from the Units view.

Code Import is the process of reverse engineering an Object Pascal source file into entities that make up the *Code Model*. This can be initial importing - the unit and / or classes it contains did previously not exist in the Code Model - or ***refresh import*** - re-importing an existing unit and / or the classes it contains in order to synchronize the Code Model and source file. Code Import is typically controlled from the Units view. It is amongst others also available in the classes view and difference view. Code Import is described in detail in chapter Importing Source Code, page 56

Design patterns are proven solutions for a general design problem. It consists of communicating classes and objects that are customized to solve the problem in a particular context. In ModelMaker patterns are active agents that will insert code into the model and stay *alive* to *reflect changes* in the model to the pattern related code. Design patterns currently only relate to the *Code Model*. Patterns are manipulated in the Patterns view. There is more on patterns in the Design Patterns manual.

Code Templates are user definable and parameterizable snippets of related code. They are like user definable patterns. Code templates can be created form the Members view and applied from the Patterns view or Members view. There's more in Code Templates page 71

ModelMaker's ***Delphi IDE integration*** takes care of synchronizing the Delphi IDE editor buffers whenever a file is (re-)generated by ModelMaker. Depending on the Delphi version there are other functions available like: add IDE editor file to model, refresh IDE editor file etc. There's more in chapter Integration with the IDE page 105

The ***ModelMaker Code Explorer*** is a separate *ModelMaker Tools* product that brings basic ModelMaker Code Model related functionality into the Delphi IDE. With this explorer you can navigate and add, edit, copy properties, methods or even entire classes with the same ease and concepts as in ModelMaker, usually even with the same dialogs.

A ***Macro*** is a fragment of text that is identified by a macro identifier. While generating source code and in-source documentation, ModelMaker will expand the text, replacing macro identifiers with the macro's text. Macros are also used to customize certain parts of the generation process (custom class separator, method section separator etc.). Macros are maintained in the Macros view. Macros are described in detail in chapter Macros, page 74 Macros are also used to parameterize *Code Templates* as described in chapter Code templates, page 71

All Code Model entities and Diagram symbols can be documented with ***Documentation*** and a ***One Liner***. In all relevant editors you'll find a Documentation tab. The Documentation view and Floating Documentation view are dedicated to editing One Liners and Documentation. Emitting "in-source documentation" during code generation is controlled by macros. You can redefine these macros to customize the documentation format.

Documentation can be converted to a unit based ***Help File***. This is done in the Documentation view. Other documentation output formats can be created with (third party) plug-in experts that use the ***MMToolsApi*** to access the model.

The ***Diagrams*** contains multiple types of diagrams. Some diagrams visualize aspects of the Code Model in UML-style. Others visualize entities such as Use Cases that only exist in Diagrams. Most symbols can be ‘HotLinked” to entities in the code model. Class symbols for example are linked strictly to classes in the Code Model: changing the class in a diagram will also change the class in the code model. Messages in sequence diagrams can be weak linked: they can or cannot be linked to a class member. If they are not linked the message name is just text. Symbols such as Use Case symbols only exist in the Diagram model and have usually no relation with the code model. It is important to realize that symbols linked to the code model (for example Class symbols in class diagrams) only *visualize* an entity (class) in the code model. The same class can be visualized many times in multiple diagrams in different styles depending on the context. Diagrams are created and maintained in the Diagram list view. The actual diagrams are edited in the Diagram Editor view. There’s more in chapter Diagrams, page 39

Diagrams can be ***exported*** as image, native XML format or in XMI format (third party plug-in expert).

The Environment and Project options are a first means to ***Customizing*** ModelMaker to your taste or coding style.

In Chapter “Customizing” page 80 there’s more on customizing ModelMaker.

ModelMaker ***Projects*** or ***Models*** contain both Code Model and Diagrams. In ModelMaker you work on a single model at a time. Opening a different model will close the model you were working on.

Working with models requires care in the areas ***team development*** and model (system) ***boundaries*** as described in chapter Team development. Model boundaries and Version Control, page 43

The ***Difference*** view is used to compare a model unit to the associated file on disk. Most powerful is the structured difference that does a syntactical comparison rather than a plain file based comparison. Also use the Difference view to compare any disk file or model unit with any other file or model unit or to compare two classes.

Code Model contents

A ***Class*** is the most important entity in the Code model, it matches the corresponding concept in Delphi and it is a container of Class Members. Classes always have an ancestor (super) class and sometimes have descendent (sub) classes. The default class ancestor `TObject` is always present in the model.

An ***Interface*** is similar to a class, as it matches the corresponding concept in Delphi. It is also a container of a restricted set of Class Members. Interfaces always have an ancestor (super) interface and sometimes have descendent (sub) interfaces. From Delphi 6 onwards there are two interface roots; IUnknown and IInterface. These default interface ancestors are always present in the model. Because classes and interfaces are so similar, in the remainder of this manual usually where you read class you can also read interface. Both are

Class Members are the fields, methods, properties and events making up a class's interface. They always belong to a class (or interface).

Fields, Methods and ***Properties*** match the corresponding concepts in Delphi. Fields are used to store a class's state and/or data. Methods are used to implement behavior. A method's implementation consists of ***sections*** of code. This allows ModelMaker to locate specific code within the method's body. Properties let you have controlled access to a class's attributes as though they were fields. ***Events*** are a special kind of properties. They are used to represent method pointer type properties (delegates). This way ModelMaker makes the same distinction as Delphi's Object Inspector does. It is possible to create a *property* of type `TNotifyEvent` and ModelMaker will generate the correct code for the property, but ModelMaker will not recognize this property as an event. Therefore: use Events rather than properties to model event types.

Event type definitions are used to define the signature of event type properties. ModelMaker relies on these definitions to create and update event handler methods and event dispatch methods. The most used event type `TNotifyEvent` is automatically inserted in each model. Event types are maintained in the Events view.

Design patterns and ***Units*** as described earlier complete the code model contents.

Diagrams

ModelMaker supports a set of UML diagrams:

1. Class diagram or static structure diagram
2. Sequence diagram
3. Collaboration diagram
4. Use case diagram
5. Robustness analyses diagram (not defined in the UML)
6. Activity diagram
7. State chart diagram
8. Package diagram or Unit dependency diagram, a static structure diagram, just showing unit package symbols.
9. Implementation diagrams: Deployment diagram and Component Diagram
10. Mind Map diagram (not defined in the UML)

The basic elements of diagrams are ***symbols*** and ***associations***. The meaning and attributes of the symbols and association used are according to the UML specification. This manual will not explain the meaning and details of each symbol. In the ModelMaker on-line help you'll find a short description for each symbol and association and it's attributes. There are a number of good books available on the UML. Alternatively you could download the latest version of the UML specification as available for free on the Rational web site.

The Chapter "Diagrams" contains a detailed description of organization and editing of diagrams in ModelMaker.

Working with models

In this chapter, a model is the equivalent of a ModelMaker project that contains both Code Model and Diagrams.

Model files

Native ModelMaker will save a model into set of files:

1. <model>.mpr; contains the project settings.
2. <model>.mma; contains the project related macros.
3. <model>.mmb; contains the code model data
4. <model>.mmc; contains the documentation for the model.
5. <model>.mmd; contains the diagrams.
6. <model>.mme; contains the event type definitions.
7. <model>.mmf; contains the project messages.

If you manage your source files using a version control system, you should add these model files to version control too.

However, ModelMaker is able to bundle the project files into a single project bundle: *.mpb. To enable this, check the option 'Bundle project files' in “Options|Environment|General”. This option is checked by default. If this option is checked, the file Open and Save dialogs will have the *.mpb file type as well as the *.mpr. Using single file bundles makes it easier to work with ModelMaker projects. To convert existing projects to single file project bundles, use File|Save as and manually change the .mpr extension to an .mpb for the project. In very large projects you may find that saving and loading bundles takes some more time than the multi file projects. When using project bundles it is sufficient to add the <model>.mpb file to version control.

ModelMaker cooperates with version control systems by not allowing you to save a project that exists with read-only file attributes. Note that only the file <model>.mpr or .mpb is checked for read only attributes. In an unbundled project file the *.mma.*.mme files are not checked.

Model templates

As you probably have noticed (for example in the Getting Started demo), one of ModelMaker’s powerful features is that it’s easy to override methods and properties and that changes are automatically propagated down the inheritance tree. But to let this work, the ancestor class and the methods and properties to override must be part of the model. So it is important with which (new) model you start.

Now you may ask: why didn't I get the complete VCL as default model? This would contain all classes I ever need! The answer is that this would result in very large models, through which it is hard to navigate. We have been working with ModelMaker for quite a few years now, and it shows that it is most practical if models contain classes of a single domain only. Classes contained in a single component package typically reside in single model too. Generally this result in models typically containing 5 to 20 classes or may be up to 50 for really large models.

And that's where you need templates. Templates are just ordinary models containing classes for a certain domain that you use to derive new classes with in a certain domain. Since it's possible to have as many templates as you like, you can create nice compact templates for each relevant sub-domain: like a template for creating simple components, one for simple TCustomPanel descendants etc.

You load a template by selecting 'File|New from template' which will load the template model and then reset the model's name to 'untitled'. Any model can be used as a template, but by design ModelMaker looks in the `[installdir]\TEMPLATE` folder for template models.

ModelMaker has one special template that it uses as default. This is the model `[installdir]\DEFAULT.mpb`. You may open this model and change it to your needs or overwrite it with another template model.

Editing a model

To edit a model, ModelMaker has multiple views on the model. Most of these views are interlinked: selecting something in one view will show related information in another. Interlinking is based on:

- The current *class*, selected in the Classes view or Diagrams view. For example, the Class Members view displays the members of the current class.
- The current *class member* (if any), selected in the Class Members view.
- The current *method* (if any), equal to the current class member if that is a method. For example, the Method Implementation view displays the implementation of the current method.
- The current *unit*, selected from the Units view.

The other view like the Macros view and the Event Library view are (more or less) independent from these selections.

In the Environment options Navigation tab you'll find options to synchronize and activate views on certain events.

Ownership in ModelMaker

ModelMaker assigns an *owner* to each entity in the model. Entities can be anything from classes to methods or a section of code in a method. The owner of an entity created the entity and has exclusive rights to update or delete it whenever suitable. Usually you, ‘User’ will be the owner since you create most classes, class members etc. ModelMaker does not discriminate between users: it does not remember that John created this class and Mary that property.

Deleting an entity will automatically delete all entities it owns too. For example: a property owns it’s read access method. You cannot delete or edit these methods, other than by deleting or editing the property.

Team development, Model boundaries and Version Control

In ModelMaker team development support and model boundaries are related issues as they both deal with the question: what should and what should not be in a model. There are a few important reasons to use multiple relative small models according to logical boundaries rather than one big model containing all classes and diagrams you have.

1. It enables team development: while one developer works on one model that is part of a larger project, the other can work on another. There are limited possibilities to merge changes made to the same model. Therefore only one developer can work on a single model at the time.
2. It improved ease of navigation and overview
3. It improves performance.

Usually you'll find logical boundaries to split up models - and usually well-designed modules (as in units or classes) have low coupling and dependencies. Boundaries could be: units in a certain Delphi package, units containing classes that perform a related task etc. In ModelMaker itself for example, we have lots of models, sometimes only containing a single unit. Large models for example contain all classes related to diagrams, the entire code engine or the source importer. And yet other models contain units that are used in many projects: timers, filters etc.

Although we have some happy customers that have models containing 300+ classes in 150+ units, practice shows that a good model size is about 1 to 30 classes in 1..10 units.

The drawback on having multiple smaller models is that ModelMaker maintains active relations only within the same model and not beyond model boundaries. As a result you occasionally might need to re-import a unit after it has been altered in another model.

It’s a good idea to use a version control system and put the model files under version control too. Although a model contains everything that is needed to (re-)generate the source files it contains, you should ***always*** store the actual source files under version control too. It's from the source files that you build your product, not the model.

Merging models is only partly supported: source code can be generated and imported and diagrams can be exported/imported. However no code model meta-information can be exported/imported. Because ModelMaker stores its data in a native binary format, you cannot use the merging capabilities of a Version Control System without corrupting the model.

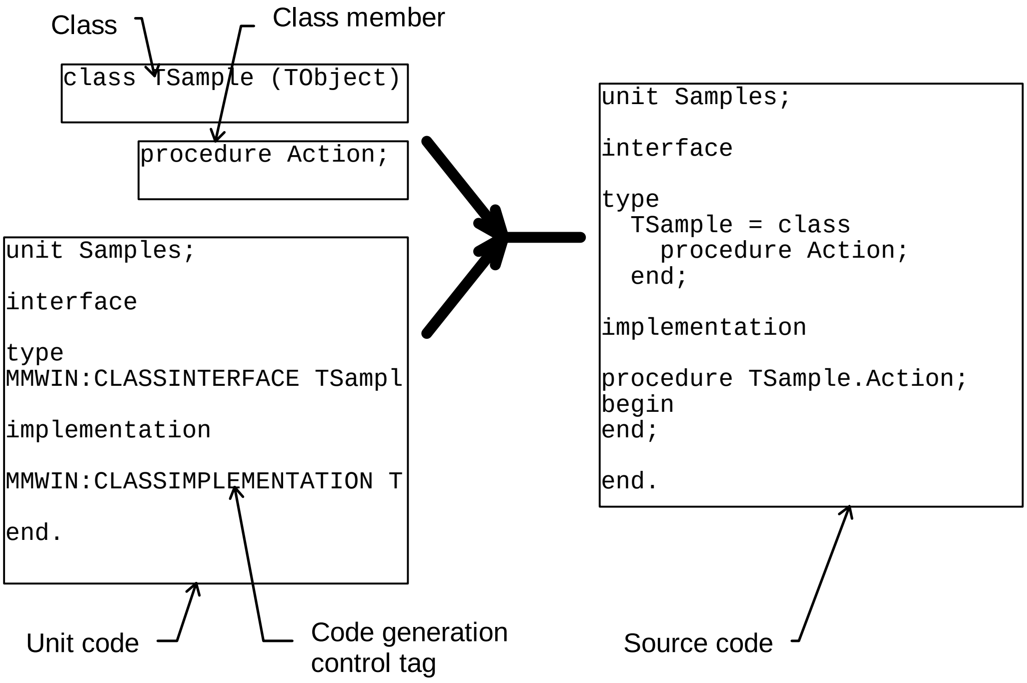
If models do get out of sync with the source code or other models you've always got the Difference View with it's powerful structural difference function to help getting the model synchronized with source or other models.

Using source aliases rather than hard coded directories is a must in team development. Check chapter Source Aliases, page 52 on source aliases.

Generation source code

Overview

In ModelMaker units are the gateways to source files on disk. During code generation, unit code, classes and class members from the code model are combined into a source file. A unit’s unit code contains (can contain) code generation control tags. At the position of a tag ModelMaker will insert the associated entity such as class interface or implementation. Here is a picture that visualizes this process.



```
class TSample (TObject)
```

```
procedure Action;
```

```
unit Samples;

interface

type
MMWIN:CLASSINTERFACE TSampl

implementation

MMWIN:CLASSIMPLEMENTATION T

end.
```

```
unit Samples;

interface

type
  TSample = class
    procedure Action;
  end;

implementation

procedure TSample.Action;
begin
end;

end.
```

The unit code is read line-by-line and scanned for code generation control tags. If a line contains a code generation control tag, the entity as defined by that tag is inserted instead of the tag. Any lines not containing code tags are just copied to the source file.

During code generation macros in both unit code and method implementation code will be expanded using the predefined macros and the project and environment macros you define yourself. Macro expansion and line formatting are the last stages in the source code generation process for both text generated from code tags and text just copied from a unit’s unit code. Macros are explained in detail in chapter Macros, page 74

During generation of the implementation section, class separators, method separators and method section separators can be emitted. This is controlled by code generation settings and macros.

Insertion of in-source documentation can also be part of the generation. This is explained in detail in chapter “In source Documentation” , page 67.

Code generation control tags

ModelMaker uses code generation control tags to control source file generation. Only code generation control tags placed in the unit code are interpreted. Tags in a method’s implementation are not interpreted. Here are the rules that apply to code generation control tags:

1. Code generation control tags are *case insensitive*.
2. All code generation control tags start with `MMWIN:` at the *first* position of a line.
3. Code generation control tags must reside on a *single line*.
4. Any *semi-colons* or *equal* signs defined in a tag are obligatory.
5. Code generation control tags can contain any white space after the `MMWIN:` definition. For example: `MMWIN:STARTINTERFACE` is the same as tag `MMWIN: START INTERFACE`

Class related tags

These code generation control tags are used to define the insertion position of class and interface related code:

```
MMWIN:CLASS INTERFACE classname ;ID=###;
MMWIN:CLASS IMPLEMENTATION classname ;ID=###;
MMWIN:CLASS REGISTRATION classname ;ID=###;PAGE=vcl page name
MMWIN:CLASS INITIALIZATION classname;ID=###; // OBSOLETE
```

These class related tags are automatically inserted and maintained by ModelMaker whenever you add a class to a unit or remove it again. Normally you would use the Unit editor dialog or drag and drop in the Units view to insert or delete classes in/from a unit or change the relative position within a unit. However, in special cases you can manually move these tags to any other position in the unit code. This is for example useful if you want the interface of a class to reside in the unit’s implementation.

In these tags ModelMaker ignores the ‘`classname`’ and just uses the ‘`ID=###`’ tag to identify the class. The class name is inserted just for your convenience. Modifying it will have no effect.

Normally, for classes both the `CLASS INTERFACE` and `CLASS IMPLEMENTATION` tags should be put in the unit code. If either one is missing after you’ve edited the unit code manually, you’ll get a warning. In special cases - for example in include or documentation files - you may manually remove either the interface or implementation tag. Interfaces (as opposed to classes) ignore the `IMPLEMENTATION` tag.

The `CLASS REGISTRATION` tag is optional and is used to insert a snippet of code to register the class as component. You may manually remove them from the unit code. You can manually edit the 'vcl page name=...' text in the registration tag to change the VCL registration page. However, usually you would do this in the unit editor dialog. If you remove the registration tag or the page name, no registration code will be generated for the class.

When importing a unit containing a procedure Register, the registration code is automatically converted to tags.
The tag CLASS INITIALIZATION is obsolete from version MMv6.0 onwards. This tag is maintained for backward compatibility only. The tag is used to insert initialization code for TStreamable descendants: RegisterStreamable(..); If you remove the initialization tag, no initialization code will be generated for the class.

Event type declaration tag

This code generation control tag is used to define the insertion position of an event type declaration.

```
MMWIN: EVENT DEFINITION eventname type declaration; ID=###;
```

This declaration is maintained by ModelMaker and is obligatory for each event definition in a unit. Normally you use the unit editor dialog or drag and drop from the Events and Units view to insert or remove event type definitions in/from a unit, or change their relative positions within a unit. However, you may manually move these tags to any other position. If you remove them, the event type definition is also removed from the unit. The 'eventname' and 'type declaration' texts are ignored, only 'ID=###' is used to identify an event type definition.

Editing marker tags

These code generation control tags are used to mark the positions at which you want ModelMaker to insert the first class or event type declaration in a unit.

```
MMWIN:STARTINTERFACE
MMWIN:STARTIMPLEMENTATION
```

These tags are for editing purposes only and they have no role in the code generation process. There's one exception to this. The MMWIN:STARTINTERFACE tag is also used to determine the insertion position of class forward declarations - if any. If this MMWIN:STARTINTERFACE tag is absent, class forward declarations will be inserted before the first event type declaration or class interface, which ever comes first.

Macro expansion control tags

These code generation control tags are used to switch on and off macro expansion during code generation:

```
MMWIN:START EXPAND
MMWIN:END EXPAND
```

By default the expansion is switched ON. You need these tags if your unit or method code contains the text “`<!`” (“” not included). The macro expander will interpret the sequences

```
“<!” + Identifier + “!>” on a single line
“<!” + Identifier + “(“ param list + “)” + “!>”
```

as a macro. “Identifier” (the macro name) can consist of characters ['0'..'9', 'a'..'z', 'A'..'Z', '_']. Which is similar to Object Pascal identifiers, although macro names can start with a number. White space surrounding the identifier is ignored.

Because macros can be in any text including comments and strings, this would make it impossible to generate code for units that contain a valid macro sequence <!ident!>.

There are a few workarounds for this problem. The most sensible uses these control tags in the unit code - remember the tags do not work in method code!

```
MMWIN:ENDEXPAND
const
  HTMLCommentStart = ‘<!’;
  HTMLCommentEnd = ‘!>’;
MMWIN:STARTEXPAND
```

In the rest of the code you can now use the constants and still leave the macro expansion on.

Note that these tags do not affect generation of in-source documentation, which is also based on macros. Documentation related macros are always expanded regardless of the setting of these switches.

Unit documentation tag (obsolete)

This code generation control tags defines the insertion position of a unit’s documentation. Because inserting unit documentation is much better controlled with the ‘In-source documentation generation’ options, we recommend avoiding the use of this tag although it’s still supported for backward compatibility.

```
MMWIN:INCLUDE UNITDOC;INDENT=##;
```

The tag ‘`INCLUDE UNIT DOC`’ defines the position at which ModelMaker will insert the unit’s documentation. The unit’s documentation can be edited in the Documentation view. The ‘`INDENT=##;`’ extension is optional and may be omitted. This defines an indention for the documentation of `##` spaces. A typical use would be:

```
{
MMWIN:INCLUDE UNIT DOC;INDENT=2;
}
unit <!UnitName!>
```

Obsolete tags

These code generation control tags are obsolete from MMv6.0 onwards.

```
MMWIN:STARTREGISTRATION
MMWIN:STARTINITIALIZATION
```

On loading a model, ModelMaker will remove these tags from the unit code. If you add them manually they will simply be ignored.

Code generation options

In the Project options|Code Generation tab you will find options that control code generation:
1. Formatting the layout of source code
2. Sorting of class members and method implementations
3. Inserting a (custom) class header to that precedes the class’s method implementations
4. Inserting a (custom) method separator to that precedes each method’s implementation
5. Inserting a (custom) method section divider in between each section.

The on-line help file explains the meaning of these options.

The Project options|Source Doc generation tab controls the generation of in-source documentation. This is explained in detail in the chapter on “In source Documentation”, page 67.

Maintaining Code Order / Custom member order

ModelMaker supports a user definable custom member interface and method implementation order.
These custom orders can be assigned during import used during code generation. When that is done, the effect is that ModelMaker will maintain the imported code order during generation.

This is how it works:
In the Project options Code Generation tab you define a member sorting scheme in class interface and implementation generation. In these sorting schemes Custom orders can be used as a grouping or additional sorting property.

When ***grouped*** on custom order, members will be sorted according to the (original) custom code order. Members that have been added later with an unspecified custom order, will be placed after all members with a specified order.
When using the custom order to perform ***additional sorting***, the default sorting scheme will be applied and within each 'section' (visibility etc.), the custom order will be applied.
If class members have an unspecified order (which is the default for new members), the effect of enabling Custom Order during Generation or in the Members list is null.

Custom orders can be assigned during Import. In the Project options|Code Import tab you’ll find settings to enable / disable assigning the custom order during import. The import dialog allows temporarily overruling these settings.

If the Importer assigns custom orders and Generation uses grouping on Custom Order ModelMaker will effectively maintain the original code order during ’refresh import’ and append new members at the end.

Additionally Custom orders can manually be defined - per class - with the "Members custom order" dialog or with the Members view ‘Rearrange mode’.

The Rearrange dialog is available from the Members and Classes view 'Wizards' local sub menus and from the Units view 'classes' local sub menu. In this dialog you either you drag and drop to rearrange a class interface and method order or use one of the predefined sorting schemes. The dialog can be also used to clear an interface or implementation custom order. Note that manually defining a custom order will erase an imported code order.

The Members view has a 'Rearrange mode' (members local menu). In this mode an interface custom order can be assigned using drag and drop in the members view itself. For method implementation order you must use the Rearrange dialog. In the in the rearrange mode the Members view filter settings (visibility, type, category) are ignored to make sure all members are displayed. Also Members view grouping is implicitly set to "Custom Order" and sorting is predefined and cannot be changed. As a visual feedback, the background of the members is changed to a silver color in this mode.

Adjusting the unit template

Whenever you add a new unit to a model, ModelMaker looks for the file `DEFUNIT.PAS` in the folder `ModelMaker\6.0\BIN`. The default unit may also be (re-)defined when adding a new unit. This text file is used as a template for the newly created unit code. You may edit this file to your needs. You can use these code generation control tags to mark the insertion position for the first class ModelMaker will insert in the unit:

```
MMWIN:START INTERFACE
MMWIN:START IMPLEMENTATION
```

You might want to adjust the default unit template in order to:

- Customize the **`uses`** clauses in the unit **`interface`** and **`implementation`**.
- Add a company-defined header.
- Control macro expansion.

As an example: here’s a unit template we use for freeware units. In the chapter Macros the same template extensively using macros is shown.

```
{
  File      : <!UnitName!>
  Version   : <!Version!>
  Comment   : <!Comment!>
  Date      : <!Date!>
  Time      : <!Time!>
  Author    : <!Author!>
  Compiler  : <!Compiler!>
+-------------------------------------------------------------------------+
| DISCLAIMER:                                                             |
| THIS SOURCE IS FREEWARE. YOU ARE ALLOWED TO USE IT IN YOUR OWN PROJECTS |
| WITHOUT ANY RESTRICTIONS. YOU ARE NOT ALLOWED TO SELL THE SOURCE CODE.  |
| THERE IS NO WARRANTY AT ALL - YOU USE IT ON YOUR OWN RISC. AUTHOR DOES  |
| NOT ASSUME ANY RESPONSIBILITY FOR ANY DAMAGE OR ANY LOSS OF TIME OR MONEY |
| DUE THE USE OF ANY PART OF THIS SOURCE CODE.                            |
+-------------------------------------------------------------------------+
}

unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
MMWIN:START INTERFACE

procedure Register;

implementation

uses StrUtils, NumUtils;

procedure Register;
begin
end;

MMWIN:START IMPLEMENTATION

initialization
end.
```

Notice how this template contains (from top to bottom):

- A simple standard header and a free ware disclaimer.
- Some statistics macros, like `<!Date!>` and `<!UnitName!>`
- The basic unit structure: **`unit..interface..uses..implementation..uses..initialization..end.`**
- A macro `<!UnitName!>` to define the unit's name.
- The **`procedure`** `Register;` definition and implementation for registering components.
- A default uses clause in the unit's implementation to include some often-used units `StrUtils` and `NumUtils`.

Unit Time Stamp Checking

The ModelMaker code generator by default uses Time Stamp checking to prevent overwriting source files that may have been changed outside ModelMaker. It checks if the file on disk is newer than the last time a unit was generated. If this is the case you'll be warned that you are about to overwrite that modified file.

You can switch on and off time stamp checking in the Environment options|General tab.

There is a limitation on time stamp checking you must be aware of:

1. If you rename a unit in ModelMaker the time stamp is not reset to 'unknown', so if you have an existing file on disk which is NEWER than the last time the unit was generated (with the old name) you will NOT get a warning. The Unit editor dialog warns you for this (file xxxx already exists, overwrite?), but the in place editor in the Units view does NOT.

The Unit difference View displays the time stamp comparison on activation. The function "Check Time stamps" refreshes this comparison.

Source Aliases

ModelMaker supports source path aliases in units to avoid hard-coded directories and make your models machine-independent. An alias is associated with an aliased directory, similar to database aliases. In the model a unit's alias is saved rather than the aliased directory. On each machine aliased paths can be defined differently. This allows you to transport models to other machines.

Source aliases are a must in team development.

Example:

Suppose on machine A you have a source directory:
`C:\DATA\PROJECTS\COMMON`
An alias defining this directory could be COMMON.
On machine B, COMMON could de defined as
`\\PROJECTDATA\COMMON`

You add aliases from the main menu "Options|Source aliases", or from the Units view popup menu.

To avoid that you need to mimic a large directory structure in aliases, unit names can be relative to an alias. That way you only need to define a few aliases for root paths.

Here's an example:
Suppose you have a directory structure which looks like this:

C:\Project1\App_a\source
C:\Project1\App_a\components
C:\Project1\App_a\utils
C:\Project1\App_b\source
C:\Project1\App_b\components
C:\Project1\App_b\utils
C:\AllProjects\source
C:\AllProjects\components
etc.

You could define three aliases
App_a = C:\Project1\App_a

App_b = C:\Project1\App_b
AllProjects = C:\AllProjects

A unit named C:\Project1\App_a\source\Samples.pas could then use

Alias = “App_a” (omit the “”)
Relative unit name = “source\Samples.pas” (omit the “”)

Source code aliases are also used to offset the Import source code dialog's initial directory. If you define an alias VCL Source for C:\Program files\Borland\Delphi 3.0\Source\Vcl, the import dialog can be offset to this directory by simply selecting this alias from the drop down menu.

Source aliases also participate in Version Control integration. See next chapter.

Version Control support and Aliases

By using a plug-in VCS Expert you can add Version Control capabilities to ModelMaker. Check the ModelMaker Tools web site for ready available third party VCS Experts or create your own using the MMToolsApi VCS interface.

If a VCS expert is installed, in the units view popup menu and main file menu VCS related menu items are available to manually check-in/out a model or unit. Also each time a read-only unit is about to be generated an attempt is made to check the unit out (after your confirmation). VCS Experts can add more VCS related commands to the popup menu such as ‘Add to project’, ‘History’ etc.

Usually VCS systems usually need a VCS project name to perform an operation. In ModelMaker Source aliases are used to store the user definable VCS projects.

Each source alias can (but does not need to) store a Version Control project. This string is passed on to an installed VCS integration expert whenever a VCS file related action is performed.

If you install a VCS expert in ModelMaker, the ModelMaker IDE integration experts will use the same expert to integrate VCS in the Delphi IDE.

For details, refer to your VCS system and MM VCS expert provider.

Using ModelMaker to generate Instrumentation code

ModelMaker supports generating Method instrumentation. This feature makes ModelMaker suited for generating instrumentation code for CodeSite, GpProfile and other tools instrumenting source code on a method base for profiling, tracing etc.

Method instrumentation generation is controlled with the option 'Instrumented' in the Method editor dialog. If this option is checked instrumentation code will be generated for the method. Instrumentation can be (de-) activated for all units at once with "Project options|Code generation|Instrumentation". The actual instrumentation code is defined by two macros you must add manually to the environment or project macro list: “MethodEnterInstrumentation” and “MethodExitInstrumentation” (omit the “”). These macros are expanded before the first and after the last section in a method's main body. For example (note the use of the predefined macros ClassName and MemberName in these macros)

macro MethodEnterInstrumentation=

```
CodeSite.EnterMethod(’<!ClassName!>.<!MemberName!>’);
try
```

macro MethodExitInstrumentation=

```
finally
  CodeSite.ExitMethod(’<!ClassName!>.<!MemberName!>’);
end;
```

When this is applied to:

```
procedure TSample.DoSomething;
begin
  ShowMessage(’Doing something’);
end;
```

This will result in the following code:

```
procedure TSample.DoSomething;
begin
  CodeSite.EnterMethod(’TSample.DoSomething’);
  try
  ShowMessage(’Doing something’);
  finally
    CodeSite.ExitMethod(’TSample.DoSomething’);
  end;
end;
```

To avoid creep when re-importing instrumented methods you can use the code remove tags. Check chapter “START and REMOVE tags, page 59 for details:

```
MMWIN:>>STARTREMOVE
MMWIN:>>ENDREMOVE
```

These tags may be part of a comment. Depending on the setting in the Project options|Code import tab the importer will filter out any code in between a start remove / end remove pair.

The macro MethodEnterInstrumentation using these tags would for example look like:

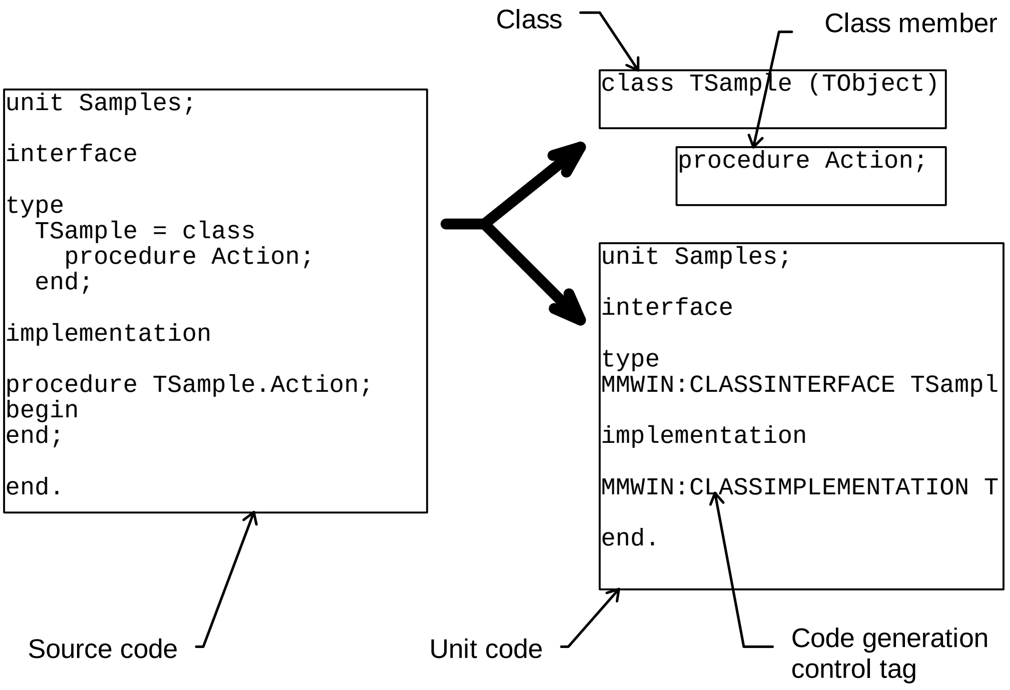
```
//MMWIN:>>STARTREMOVE
CodeSite.EnterMethod('<!ClassName!>.<!MemberName!>');
try
//MMWIN:>>ENDREMOVE
```

The Member Manipulator can be used to switch on and off Instrumentation for multiple methods at once. On the ModelMaker Tools web site you'll find a third party plug-in Instrumentation expert. This expert is dedicated to controlling method instrumentation code.

Importing source code

Background

ModelMaker imports Delphi Object Pascal source files. This process is basically the inverse from generating a source file from classes, members and unit code. The class related code is converted into Code Model entities such as classes, interfaces, members and method implementations. All non-class or interface related code is moved into a ModelMaker unit's unit code.
This process is called reverse engineering.



```
unit Samples;

interface

type
  TSample = class
    procedure Action;
  end;

implementation

procedure TSample.Action;
begin
end;

end.
```

```
class TSample (TObject)
```

```
procedure Action;
```

```
unit Samples;

interface

type
MMWIN:CLASSINTERFACE TSampl

implementation

MMWIN:CLASSIMPLEMENTATION T

end.
```

It is important to realize that if an imported class is not currently existing in the model, members and code sections are inserted as 'User created/owned" and no attempt is made to extract meta information, such as applied patterns, inherited calls etc. The only exception to this, are the read and write access members of properties, which are restored and linked to the property. For example: all code inserted by patterns is read back, but marked as "user" and the patterns itself is not recreated. The same applies for inherited method calls etc. In general you loose the meta information.

However, if an imported class already occurs in the model it is 'refreshed' and all meta-information is restored. Even applied design patterns can be traced back.

Therefore importing source code is great for:

- Importing (an interface of) a class you want to inherit from or use as a client.
- Importing existing code originally not developed with ModelMaker
- Updating an existing class from source code

However it is advised that once imported, you keep editing your code in ModelMaker. The only exceptions to this are form and (other resource module's) source files which by their nature you (partially) need to edit in Delphi.

Importing a source file

The main toolbar and units view pop-up menu contain buttons and commands to

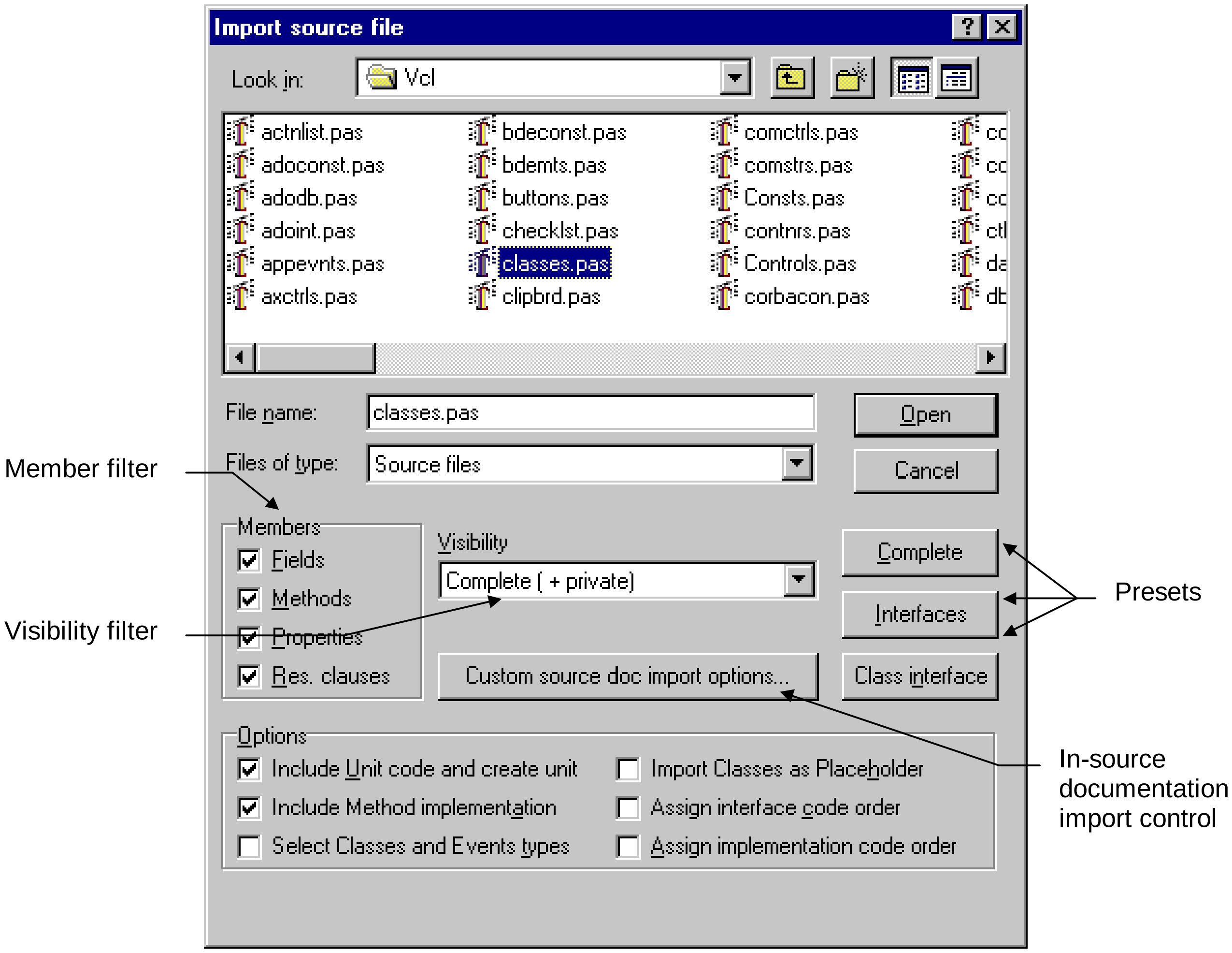
- Import a source file (in the current model or a new model).
- Refresh import source file.

Additionally you'll find import functions in the Classes view (refresh class or associated unit) and the Difference view (refresh unit, class or method).

In the Project options|Code Import tab you'll find the options to control source code import. Check the on-line help file for a detailed description on each option. The project options are used for all (refresh) imports except when you use the Import dialog to interactively import a file. The import dialog allows temporarily overruling some project import control options. The import dialog has some other options that by their nature are not in the project options, typically related to initial first time import. The options in the import dialog are preset (each time!) to the project options.

The import dialog's initial directory is pre-set by selecting a source code alias. You'll find more on (defining) Source code aliases on page 52.

When importing a source file using the import dialog, you enter a source file name and set filters and options to control the import. Most options are rather self-explanatory.



Import source file

Look in: Vcl

File name: classes.pas

Files of type: Source files

Members

☑ Fields

☑ Methods

☑ Properties

☑ Res. clauses

Visibility: Complete (+ private)

Options

☑ Include Unit code and create unit

☑ Include Method implementation

□ Select Classes and Events types

□ Import Classes as Placeholder

□ Assign interface code order

□ Assign implementation code order

If the option “Include Unit code and create unit” is checked, not only the classes will be imported, but also the non-class related unit code. Check this option if you want a complete import. If you just need a class’s interface this option can be unchecked.

If the option “Select Classes and Events to import” is checked, you may select which of the classes or events found in the source file will actually be imported in the model. Useful if you’re not interested in all classes and events contained by a unit.

If the option “Import Classes as Placeholder” is checked all imported classes will be marked placeholder. If it is unchecked classes will remain their current state or ‘real’ if not found in the current model.

There are three pre-set buttons, which set the filters to a “complete”, “interfaces only” “class interfaces” mode. The default settings (as displayed) are those for a complete import.

The In-source documentation import control settings are explained in detail in chapter “Importing a source file” page 57.

Importing (adding) versus Refreshing

‘Refresh Import’ and Import as in ‘Add to Model’ act similar - the difference is how existing units and classes are treated.

Importing with the ‘Add to Model’ function (in the IDE or ModelMaker) will add non-existing units or classes contained in the added unit. If either unit or class already exists in the model it will be refreshed.

‘Refresh Import’ issued from the IDE integration expert will only refresh the unit if already existed in the model and will not add a unit not currently in the model. In ModelMaker you can only refresh an existing unit.

Avoiding creep - removing code during import

ModelMaker supports removing certain fragments of code during import to avoid creep in a full generation / re-import cycle.

ModelMaker generated default class separators such as

```
{
***************************** TSample *****************************
}
```

and ModelMaker generated default category markers such as
{<<Category>>: Model linking}

are automatically removed.

Additionally ModelMaker will remove the following code:

1. All code marked by a MMWIN:>>STARTREMOVE and MMWIN:>>ENDREMOVE pair.
2. All comments with the ‘removal signature’ as defined in the Code Import options.
3. Matched (and optionally unmatched) comments according to the documentation style comment.

For importing “In source documentation” and removing comments with the documentation signature(s), refer to chapter on “Importing in source Documentation”, page 68.

STARTREMOVE and ENDREMOVE tags

You can use these tags in any code, comment or string to instruct ModelMaker to remove all code between the tags including the tags themselves:

```
MMWIN:>>STARTREMOVE
MMWIN:>>ENDREMOVE
```

The corresponding setting in the Project options|Code import tab will enable/disable filtering based on these tags.

These tags are commonly used to remove (part of) a macro that was generated by ModelMaker from the input file. Check the chapter on generating Method Instrumentation code for an example.

Comments with remove signature

If you want to remove certain comments from the source file you can use comments with the Removal Signature. This signature is defined in the Project options|Code import tab which also enables and disables removing comments with this signature.

You will need to use this type of comment remove filter if you
1. Define a custom class separator,
2. Define a custom method separator,
3. Define a custom method section separator
4. Redefine the category expansion macros 'IntfCategory' or 'ImplCategory'

Assuming {- } to be the removal style comment, a custom class header should look like

```
{-
****************************
*
*           <!Classname!>
*
****************************
}
```

Similar, a custom category expansion tag could look like
`{- Category: <!Category!> }`

Note that Method End Documentation is automatically removed due to the fact that the importer will remove the method including the line containing the method's final **`end;`**.

Import restrictions and limitations

ModelMaker usually imports in about 99.9 % of all cases without problems. If ModelMaker generated the source file, the imported code is usually 100 % correct. ModelMaker uses a combination of syntactical analyses and line based extraction to support importing of code that is not entirely syntactically correct.

ModelMaker's import mechanism imposes the following restrictions on source files in order to be imported correct. ModelMaker's importer uses the same parser as the ModelMaker Code Explorer. The ModelMaker Code Explorer that integrates in the Delphi IDE will display a list of parse errors. That way it acts more or less as tool to check code before importing into ModelMaker.

Class and Interface interfaces

Restrictions in class and interface declarations.

Any comments, compiler directives and white space in a class's interface are/is ignored except in method parameter lists.

Comma separated Field declarations are converted into separate fields:

```
FA, FB: Integer;
```

is imported as

```
FA: Integer;
FB: Integer;
```

Procedure or method pointers that are not defined as an type are not imported correct. The work around is to use a type definition.

The following code causes import errors.

```
TSample = class(TObject)
  FEvent: procedure of object;
end;
```

Which can be replaced by this code that will import correct:

```
TMyEvent = procedure of object;

TSample = class(TObject)
  FEvent: TmyEvent;
end;
```

Method implementation

Restrictions in method declarations.
Any comments, compiler directives and white space in are/is ignored except in parameter lists.

Local variables immediately following the method declaration will be converted to ModelMaker method variables. If local variables for example are preceded by a **`type`** or **`const`** declaration, they will be added to the method's local code section, just like all other local code for that method.

In the following example the local vars. I, J and S will be converted to ModelMaker local vars., the const declaration and procedure CheckIt will be placed in the method's local code section.

```
procedure TSample.Action;
var
  I, J: Integer;
  S: string;
  const CheckSum = $AAAA; // this will go into local code
  procedure CheckIt;
  begin
  end; // this is the end of the local section
begin
  CheckIt;
end;
```

In the following example the local vars. I, J and S will be placed in the method's local code section together with the const declaration:

```
procedure TSample.Action;
const CheckSum = $AAAA; // this will go into local code including the vars
var
  I, J: Integer;
  S: string;
  procedure CheckIt;
  begin
  end; // end of local code section
begin
  CheckIt;
end;
```

During refresh import of a method already existing in the model, the importer will leave the code sections intact wherever possible. If the importer cannot locate a non-user owned section of code, it will simply leave the section in the method and give a warning.

Comments and white space

The following table shows how ModelMaker treats comments and compiler directives

Comment or compiler directive in:	Import result
Class interface	Ignored
Method header	Ignored except in parameter list
Local vars.	Ignored
Method local code and body	Copied to method
All other code	Copied to unit code

Check the ModelMaker generated default class separators such as

```
{
***************************** TSample *****************************
}
```

and ModelMaker generated default category markers such as
{<<Category>>: Model linking }

are automatically removed.

If you define a custom class separator, method section separator, or redefine for example the category expansion macro TODO, you should use the remove style comments to avoid creep during import. Check paragraph " Comments with remove signature", page 60.

Unsupported language constructs

Include files are not read during import, so if you find yourself thinking: "where's my method implementation gone?" you probably need to add the included files to the imported unit using a text editor and re-import the file.

Compiler directives in class interfaces are not supported and are a potential problem source. Using inheritance may sometimes solve this. Worst case you need to create two units.

Compiler directives *around* method implementations are not supported. Placing the directives inside the method can solve this:

```
{$IFDEF DEMO}
procedure TSample.Action;
begin
end;
{$ELSE}
procedure TSample.Action;
begin
  { actually do something useful }
end;
{$ENDIF}
```

Won't be imported correct, but can be replaced by the following code which will be imported fine:

```
procedure TSample.Action;
begin
{$IFDEF DEMO}
{$ELSE}
{ actually do something useful }
{$ENDIF}
end;
```

The importer matches **`begin..end try..end, case..end`** pairs etc. to locate methods. Because conditional defines are not interpreted, using conditional defines you can create code that will compile correct but will not import correct. In fact The Delphi IDE background compiler uses a similar mechanism and will not be able to function properly either when inserting new methods in code completion or creating a new event handler.

This code for example will confuse the importer's begin end matching. The method will not be imported correct.

```
procedure TSample.Action;
{$IFDEF DEMO}
var
  S: string;
begin
  S := 'Demo';
  ShowMessage(S);
{$ELSE}
begin
{$ENDIF}
end;
```

You can replace the previous code by the following code that will import correct and as a side effect allows the Delphi IDE to stay on track too:

```
procedure TSample.Action;
{$IFDEF DEMO}
var
  S: string;
{$ENDIF}
begin
{$IFDEF DEMO}
```

```
S := ‘Demo’;
  ShowMessage(S);
{$ENDIF}
end;
```

Pure assembler methods are not supported:

```
procedure TSample.Fast; assembler;
asm
end;
```

Expressions in an indexed property’s index specifier are not supported:

```
property FirstPicture: TBitmap index BM_USER + 0 read GetPicture;
property SecondPicture: TBitmap index BM_USER + 1 read GetPicture;
property ThirdPicture: TBitmap index BM_USER + 2 read GetPicture;
```

This can be solved like:

```
property FirstPicture: TBitmap index BM_FIRST read GetPicture;
property SecondPicture: TBitmap index BM_SECOND read GetPicture;
property ThirdPicture: TBitmap index BM_THIRD read GetPicture;
```

Conversion errors

Any import conversion errors or warnings will be displayed in the Message View. The messages may be printed, saved etc.

Not reported conversion errors are:
1. Minor changes in property access method parameters lists.
2. Positioning of code generation tags in the unit code.

The best thing to do after importing a complex unit, is to perform a Delphi syntax check on the re-generated unit. From our experience it shows that if there are any remaining errors, they will evolve here.

Another option is to make a file based difference in the Difference view between the imported unit and the original source file. You should make sure that Code generation sorting scheme matches the scheme used in the original file. You might need to import Custom Code order and use the same order during generation to maintain code order. Check chapter “Maintaining Code Order / Custom member order” on page 49.

Note that to build a *structured* difference the same importer is used that will hide the same type of errors!

Auto Refresh Import

The Auto Refresh Import feature is only available together with the Delphi 4 and higher. This function will automatically refresh a unit in ModelMaker if you save the unit in the Delphi IDE. This improves synchronization of code developed both in ModelMaker and the Delphi IDE at the same time. However there are some serious warnings.

How it works

If you change a unit in the IDE editor that is also maintained in a ModelMaker model, the model and source file will be out of sync. Normally you have to 'refresh import' the unit to synchronize the model with the changes on disk. The auto-refresh import feature will do this automatically each time you save a unit in the IDE.

How it is activated and controlled

In the ModelMaker menu in the Delphi 4 (and higher) IDE check the item 'Enable Auto Refresh'. If this option is set, each time you save a unit (or project) in the IDE, the 'Auto Refresh command is send to ModelMaker which checks if:

1. The Environment option 'Auto refresh Import' is checked
2. The unit is in the current model
3. Unit generation is not (user) locked
4. The unit has 'auto code generation' enabled

If all above conditions are met, ModelMaker will do a refresh import and unlike after a manual 'refresh import' leave the unit in 'auto generation enabled' mode and - this is ***very important*** - *regenerate* the unit.

Effectively Auto Refresh improves synchronization between ModelMaker and the Delphi IDE: whenever you change something in ModelMaker, the auto-generation enabled unit will regenerate the file and reload it in the IDE. Whenever you change and save a file in the IDE, ModelMaker will resynchronize it in the model.

Warnings

In the normal, non-auto refresh development model you always have one master and slave: either ModelMaker refreshes the IDE using automatic code generation or you manually refresh the ModelMaker model with the IDE if you want to resynchronize again. With this feature there's no master or slave anymore. This can seriously damage your work as may be clear from the following example: When refreshing the unit, ModelMaker assumes it's reading a 'compilable unit'. If you for example have omitted a single begin or 'end;' or worse, comment out something, have unterminated strings etc. class and method import will be in trouble and not detect your error but simply remove all 'unwanted' methods. Since ModelMaker detected a change the unit is auto-generated and immediately after you saved the unit is reloaded with disastrous results. Experience shows that it's easy to loose lots of work instantly. Auto Refresh must be used with great care. Note that if Auto Save is enabled in the IDE, the Compile / Run command will auto save modified units depending on your IDE environment settings.

If you want more control, rather than just save in the IDE invokes auto refresh, you can use the ‘Refresh Import’ command from the ModelMaker IDE expert to save a file. This command will not only generate a manual refresh import command but also automatically save your unit in the IDE. Therefore you could use this command with shortcut Ctrl+Shift+H rather than the conventional Ctrl+S to save and refresh. You can add the ModelMaker Refresh command to the IDE tool bar. Check chapter (Integration in) Delphi IDE page 106.

Editing Form source files

ModelMaker Tools developed the ModelMaker Code Explorer to help editing Form, Data Module and other resource module source files. Due to the nature of these resource module files the IDE editor is more suitable for editing them. The ModelMaker Code Explorer will dock into the IDE editor and bring basic Code Model editing and navigation actions right into the IDE.

If you do not have the ModelMaker Code Explorer installed, you *can* edit form (and other resource module) source files with ModelMaker. This offers many advantages:

1. Use ModelMaker’s high level view and filters to navigate through your form code.
2. Automatically *restructure* your source files by regenerating them.
3. Improve your form code quality by adding methods and (array) properties with the same ease as for “normal” non form classes. Especially when you turn your forms into components, you want them to have nice and clean code to improve maintainability.
4. In general speed up form implementation.

For a smooth cooperation between Delphi and ModelMaker, stick to these rules:

In Delphi,

- Create and rename the form and unit.
- Add, delete and rename components.
- Set component properties.
- Create, rename and delete event handler methods.

Delphi adds all it’s components and event handler methods with the default visibility, so they are easy recognized in ModelMaker (use the members filter to filter out default visibility).

In ModelMaker,

1. Import your form file in an (empty) ModelMaker model.
2. Add, edit and delete all other members
3. Add additional classes to the unit.

To synchronize between Delphi and ModelMaker,

1. Regenerate the ModelMaker unit whenever you changed your code in ModelMaker. The auto-generate feature will help you doing this automatically.
2. Refresh the ModelMaker unit whenever you changed your code in Delphi, the integration experts will help you doing this automatically.

In source documentation

Overview

ModelMaker supports generating and importing "in-source” documentation. That is: the documentation and One Liner attributes of Code Model entities can be inserted during code generation and / or read (back) during import. Generation and importing of in-source documentation is controlled by the Project options tab “Source Doc Generation” and “Source Doc Import”. The Import source dialog allows temporarily overruling the import settings.

"In-source" documentation must be marked with special (user definable) documentation and one liner signature tags, more or less like in Java Doc. These signatures must be an Object Pascal comment symbol followed by one or more letters, Common used signatures are:

Documentation tags:

```
{{
{:
(**
(*:
```

One liner tags:
{1
//:
//1

Generation is internally based on macros. To customize the generated format, you can redefine these macros. To ensure a correct round trip (generation followed by a re-import) the generation and import settings must match. Normally ModelMaker enforces a correct match by using the essential import settings for generation. If you redefine the documentation generation macros, you must ensure correct matching.

Generating in-source documentation

ModelMaker supports generating and importing ‘in-source’ documentation in source files. Generation is controlled by the Project options on tab ‘Source Doc Generation’ and some macros (refer to Macros). You do not need to define these macros, as ModelMaker will on the fly insert the required macros. You can however redefine them either as environment or as project macro to customize the generated format.

Documentation macros

Macro name	Description	Example

ModuleDeclDoc	Used to expand Module (unit) documentation. Check Module related macros	`{{ module`
		`<!ModuleDoc!>`
		`}`

EventDoc	Used to expand Event Type documentation	`{{ event type`
		`<!MemberDoc!>`
		`}`

ClassIntDoc	Used to expand Class documentation in the class declaration	`{{ class <!ClassName!>`
		`<!ClassDoc!>`
		`}`

ClassImpDoc	Used to expand Class documentation in the class implementation (emitted just before the first method implementation)	`{{ class <!ClassName!>`
		`<!ClassDoc!>`
		`}`

MemberIntDoc	Used to expand member documentation in the class interface	`{{ <!ClassName!>.<!MemberName!>`
		`<!MemberDoc!>`
		`}`

MemberImpDoc	Used to expand method implementation documentation	`{{ <!ClassName!>.<!MemberName!>`
		`<!MemberDoc!>`
		`}`

MemberEndMacro	used to expand method termination documentation	`{ <!ClassName!>.<!MemberName!>`
OneLinerMacro	Used to expand one liners in classes, members, units and event types	`//: Summary: <!OneLiner!>`

In the examples it is assumed that {{ is the documentation signature and //: is the One Liner signature.

As you see in the examples, you can use any of the predefined macros such as <!ClassName!> inside the documentation macros.

However, ModelMaker is only able to import certain styles of source documentation. This is important if you want to ‘Refresh Import’ a unit.

Importing in-source documentation

ModelMaker supports importing "in-source" documentation. Importing source documentation is controlled by settings in the Project options tab “Source Doc Import” and the Import source dialog. As explained: "In-source" documentation and One Liners must be marked with special (user definable) documentation signature tags. More or less like in Java Doc.

An example:

```
//1 SomeMethod does something useful.
{{
procedure TSample.SomeMethod
This method does something useful
It would take pages to tell what.
}
procedure TSample.SomeMethod;
begin
end;
```

In the above example the OneLiner signature is defined as //1 and the documentation signature is defined as {{. The whole comment until the matching comment end symbol is treated as documentation for the first entity defined following or preceding it - depending on the documentation import options. In the example above the comment will be assigned to TSample.SomeMethod. You cannot use multiple line // style comments for documentation.

In the Source documentation options there are three documentation import modes:

1. Import: enables source documentation import and replaces documentation in the model with that in the source file,
2. Clean up: leaves the documentation in the model unaffected, but removes documentation from the unit-code
3. Inactive: which does nothing and leaves the documentation in the unit code.

The option ‘Remove unmatched documentation determines if only matched documentation should be removed or just any comment with the documentation signatures.

When assigning the comment to the Documentation attribute, the first and last #n lines are ignored. Defining any other value than 1 (one) is only useful if you redefine the documentation generation macros. The standard macros assume a value of 1.

You could use the fact that the first n lines (user definable) are ignored and place there the additional macros you'd like to generate. For example:

```
{{
<!ClassName!>.<!MemberName!>
(<!Visibility!>)
<!MemberDoc!>
}
```

This macro would require the first three (3) lines and the last one (1) to be skipped. Using this technique the member documentation won't grow each time you (refresh) import a unit. Another option is to use the Import "clean-up" import mode after in-source documentation has been imported once.

Alternatively you could leave the number of lines to be skipped at 1, and use additional removal style comments to customize your in-source documentation format.

For example:

```
macro ClassIntDoc =
{- ************
<!ClassName!>
```

```
**************}
{{
<!ClassDoc!>
}
```

Note the removal style comment in the first part of this macro. Check chapter “Comments with Remove signature”, page 60.

Related to this is the use of a custom ClassSeparator or MethodSeparator to emit more than just the documentation preceeding the class or a method implementation. Refer to Code Generation options. Using custom separators has the advantage of not needing the redefine the documentation macros and that way ensuring that all expanded documentation looks similar.

For example, defining a MethodSeparator macro like this:

```
{- <!ClassName!>.<!MemberName!>
  (<!Visibility!>) }
```

Combined with enabling the method implementation in source documentation (without redefining the related macro), this behaves as if the entire macro were:

```
{- <!ClassName!>.<!MemberName!>
  (<!Visibility!>) }
{{
<!MemberDoc!>
}
```

Which would be emitted for example like this:

```
{- TSample.SomeMethod
  (public) }
{{
This is the documentation for the method
}
```

Note the use of removal style comments for the MethodSeparator that avoids creep when re-importing the generated code.

Code templates

ModelMaker supports the use of code templates. They are like user definable patterns and consist of a (usually consistent) set of members that is put in a code template source file. This template can then be applied whenever needed again. The template file acts like a structured persistent copy / paste buffer. The powerful aspect is that templates can be parameterized using user definable macros. ModelMaker will extract the macros, let you edit them and expand the macros before importing the template file. The template files can be edited in Delphi, but should not be imported in ModelMaker directly.

Creating a Code template

To create a code template, in the Members view select the members you want the template to contain use the popup menu 'Create code template’. You'll be prompted for a file name. ModelMaker then generates the selected members as part of a stand-in class named `TCodeTemplate`. Here's an example of a simple template file containing a simple `Items` property with a standard `TList` implementation

```
unit SimpleItems;

  TCodeTemplate = class (TObject)
  private
    FItems: TList;
  protected
    function GetItems(Index: Integer): TObject;
  public
    property Items[Index: Integer]: TObject read GetItems;
  end;

function TCodeTemplate.GetItems(Index: Integer): TObject;
begin
  Result := TObject(FItems[Index]);
end;
```

Applying a Code template

To apply a previously created template, select the popup menu 'Apply template’ from the Members view. You'll be prompted for a template file name. ModelMaker will import the members contained by the first class in the template unit and add them to the currently selected class. Other classes and all other code in the unit are/is ignored. There's one exception: event type definitions can also be added to a code template, they will automatically be added to the event library when applying the template.

Registering a Code template

Code Templates can be registered on the patterns palette (patterns view) and Code Template palette (members view popup menu). These palettes that look like the Delphi component palette make it even easier to apply a Code Template. To (un)register a template use the popup menu functions in the palettes or select the corresponding option when you create a Code Template. Code Templates are shared with the ModelMaker Code Explorer.

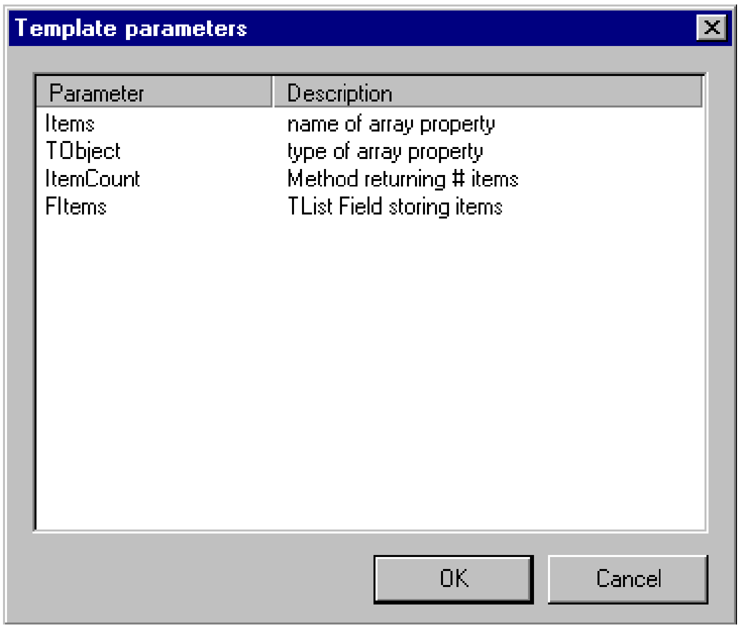
Parameterize a Code template using macros

You can parameterize a code template by adding macros to the template unit. When applying a template, ModelMaker will first extract the macro parameters, let you edit them, expand the code template and finally apply the template. This allows you to create more generic templates. A macro definition should be formatted as

```
//DEFINEMACRO:macroname=macro description
```

The standard ModelMaker macro rules apply. For example: macroname must be an identifier. Parameterizing the above example could for example be done like this.

```
unit SimpleItems;
//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

  TCodeTemplate = class (TObject)
  private
    <!FItems!>: TList;
  protected
    function Get<!Items!>(Index: Integer): <!TObject!>;
  public
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
  end;

function TCodeTemplate.Get<!Items!>(Index: Integer): <!TObject!>;
begin
  Result := <!TObject!>(<!FItems!>[Index]);
end;
```



Parameter	Description
Items	name of array property
TObject	type of array property
ItemCount	Method returning # items
FItems	TList Field storing items

If you apply this template, ModelMaker will show a dialog with the list of parameters you defined: Items, TObject, ItemCount and FItems allowing you to change them for example in Members, TMember, MemberCount and FMembers. This way the template can be added multiple times in different contexts. ItemCount is not used in this example, but in the sample code template that is shipped with ModelMaker method ItemCount is part of the template.

ModelMaker predefines one macro: “ClassName” contains the name of the class the macro is applied to. You can redefine ClassName when parameterizing a template. Use ClassName for example to create a singleton implementation macro:

```
function TCodeTemplate.Instance: <!ClassName!>;
begin
  // return the single instance.
end;
```

When applied to a class named TMySample this will expand to:

```
function TMySample.Instance: TMySample;
begin
  // return the single instance.
end;
```

Macros

Overview

A macro in ModelMaker is an identifier placed between <! and !> tags. They may also include an optional parameter list. When the macro is expanded, the macro identifier and tags are substituted by the text associated to the macro. For example macro <!UnitName!> will (in unit Samples) be expanded to the actual unit's name 'Samples'.

Macros are used in

- Code Generation.
- Customizing certain aspects of code generation such as a custom class separator, inserting categories etc.
- Generating in-source documentation.
- Parameter zing Code Templates
- ModelMaker Code editor.

During code generation ModelMaker predefines some model statistics macros at run time. Such as <!`UnitName!>, <!Date!>` etc.

You define your own macros in the Macros view. Macros are defined per desktop (environment) and per project. If a macro is both in the project and the environment macros, the project macro overrules (redefines) the environment macro.

For Parameterizing Code Templates using Macros, refer to chapter Parameterize a template using macros , page 72. To expand a Code Template the predefined macros and project and environment macros are not used.

Macros in Code generation

When generating a source file from a ModelMaker unit (refer to chapter Code Generation page 45), ModelMaker will expand macros in all text that is send to the output file. Therefore macros can be placed in any code: in unit code, in a section of a method's implementation or even in a local var definition. Macro expansion is switched on and of with the generation control tags `MMWIN:START EXPAND` and `MMWIN:END EXPAND` By default the expansion is switched ON. Check chapter "Macro expansion control tags", page 47 for an example.

When expanding a macro, first the list with predefined macros is checked, then the Project macros and finally the environment macros. If an identifier is not found, the macro text is either just removed or generation is aborted depending on the setting of '`Ignore undefined macros`' in the Project options Code Generation tab.

Predefined macros

This table shows the macros ModelMaker predefines when generating a source code file. Some macros may be redefined - especially the documentation expanders, others such as Date and ClassName are fixed.

Macro Name	Allows override	Description	Example
Generic predefined macros			
`Date`	No	Generation date, for example: `19-02-2003`	
`Time`	No	Generation time, for example: `12:34:56`	
`LineNr`		The 1-based line number in the resulting source file at which the macro is defined.	
Documentation expanders			

`ModuleDeclDoc`	Yes	Used to expand Module (unit) documentation. Check Module related macros	`{{`
			`<!ModuleDoc!>`
			`}`

`EventDoc`	Yes	Used to expand Event Type documentation	`{{`
			`<!MemberDoc!>`
			`}`

`ClassIntDoc`	Yes	Used to expand Class documentation in the class declaration	`{{`
			`<!ClassDoc!>`
			`}`

`ClassImpDoc`	Yes	Used to expand Class documentation in the class implementation (emitted just before the first method implementation)	`{{`
			`<!ClassDoc!>`
			`}`

`MemberIntDoc`	Yes	Used to expand member documentation in the class interface	`{{`
			`<!MemberDoc!>`
			`}`

`MemberImpDoc`	Yes	Used to expand method implementation documentation	`{{`
			`<!MemberDoc!>`
			`}`

Customization Macros			

`ClassSeparator`	Yes	Custom Class separator, Only active if corresponding option is activated in Project Code	`{-********************`
			`** Class: <!ClassName!>`
			`** Category: <!Category!>`
			`**********************}`

		Generation options.	assuming {- is the comment remove style
`MethodSeparator`	Yes	Custom Method implementation separator. Only active if corresponding option is activated in Project Code Generation options.	`{- <!MemberName!> -}` assuming {- is the comment remove style
`SectionSeparator`	Yes	Custom Method section separator. Only active if corresponding option is activated in Project Code Generation options.	`{------- section --------}` assuming {- is the comment remove style
`IntfCategory` `ImplCategory`	Yes	Wrappers for Category emission in class or member interface and implementation.	`{- Category: <!Category!>}`
`OneLinerMacro`	Yes	Used to expand one liners (class, member, unit event type)	`//: Summary: <!OneLiner!>}`
`MethodEndMacro`	Yes	Used to expand method end documentation. Only active if corresponding option is set in Project Source documentation Generation options	`{<!ClassName!>.<!MethodName!>}`
Entity specific Macros			
`OneLiner`	No	All entities: for Modules only during Module documentation generation.	
`OneLiner` `Category`	No	All entities: for Modules only during Module documentation generation.	

`ModuleDoc`
`ModuleName`
`ModulePath`
`Alias`
`UnitName`
`UnitPath`
	No	Module specific macros (units). Always available (not only in module documentation). The UnitXYZ macros exist for backward compatibility. Use the ModuleXYZ macros in new projects	

`ClassDoc`
`ClassName`
`TrimmedClassName`
`Ancestor`
	No	Class specific macros. Valid in class and members of that class. Valid in unit code after the first class has been generated. TrimmedClassName contains the class name with the first character (’T’ / ’t’) removed.	In unit code use this to insert auto updated global variables or class pointer types:
			
			`var <!TrimmedClassName!>: <!ClassName!>;`
			`type <!ClassName!>Class = class of <!ClassName!>;`

`MemberDoc`
`MemberName`
`Visibility`
`DataType`
	No	Member specific macros. Valid during generation of declaration, documentation and method	

		implementation code.	

`Parameters`
`CallParams`
`MethodKind`
	No	Method specific macros. Valid during generation of declaration, documentation and method implementation code.	

		Event Types use the same macros as methods. Except: Category and Visibility which are undefined	

The Documentation Expander macros are used to generate in-source documentation. To customize the generated format, you must redefine these macros, either in the project or environment macros. Refer to chapter “Generating in source documentation”, page 67.

The Customization macros can be (re)defined to customize the format of the related aspect. ClassSeparator, MethodSeparator and SectionSeparator are only active if the corresponding options are checked in the project Code Generation options. Refer to chapter “Code Generation Options”, page 49.

Using Macros in code

Some rules that apply to using macros:

- An entire macro including start and end tags must reside on a single line.
- Macro identifiers can contain characters ['0'..'9', 'a'..'z', 'A'..'Z', '_']. This is similar to Object Pascal identifiers, although macro names can start with a number.
- White space surrounding the macro identifier is ignored.
- A start tag not followed by a valid identifier is not considered to be a macro
- If the macro identifier is not followed by either the end tag or the parameter list, the macro is not considered to be a macro.

Rather than presenting the macro syntax diagram, an example will demonstrate the use and definition of macros.

Assume these (user) macros to be defined, in either the project macro list or in the environment macro list.

Name: `Author`
Parameters:
Text:

```
S.M.A.R.T. Programmer
```

Name: `Assert`
Parameters: `Cond, Msg`
Text:

```
{$IFOPT D+}
if not (<!Cond!>) then
  raise Exception.Create(’Assertion error in line <!LineNr!> of unit <!UnitName!>’ +
                         #13 + Msg);
{$ENDIF}
```

This is how you could use these macros:

```
procedure SomeAction(Index: Integer; C: Char);
begin
  <!Assert(Index >= 0, ‘Index out of range’)!>
  <!Assert(C in [‘a’, ‘b’], ‘Char out of range’)!>
  <!Assert(ValidPair(Index, C), ‘Additional checks failed’)!>
  ShowMessage(‘<!Author!> created this code’);
end;
```

Assuming the procedure was placed on line 100 in unit Demos, this text would expand to:

```
procedure SomeAction(Index: Integer; C: Char);
begin
  {$IFOPT D+}
  if not (Index >= 0) then
    raise Exception.Create(‘Assertion error in line 102 of unit Demos’ +
    #13 + ‘Index out of range’);
  {$ENDIF}
  {$IFOPT D+}
  if not (C in [‘a’, ‘b’]) then
    raise Exception.Create(‘Assertion error in line 107 of unit Demos’ +
    #13 + ‘Char out of range’);
  {$ENDIF}
  {$IFOPT D+}
  if not (ValidPair(Index, C)) then
    raise Exception.Create(‘Assertion error in line 112 of unit Demos’ +
    #13 + ‘Additional checks failed’);
  {$ENDIF}
  ShowMessage(‘S.M.A.R.T. Programmer created this code’);
end;
```

The basic rules that apply to the use of macros are:

- When using macros in text, the complete macro including its parameter list must reside on a single line. So this won’t work:
  ```
  <!Assert((A > B) and (B > C),
           ‘This is bad input’)!>
  ```
- Arguments in the parameter list are comma delimited, such as in the use of `Assert`.
- Arguments can contain even pairs of () , [] and ‘’ characters, such as in sets, arrays and string literals.
- If no parameters are defined, as in <!Author!>, you omit the brackets when using the macro.
- Macros can use other macros in their macro text. In fact even parameters can be macros. Nesting is allowed up to 15 levels.
- Circular macro definitions are illegal.
- Macros expand to plain text. See for example the use of the predefined LineNr and UnitName macros in the Assert macro’s text. The expanded macro LineNr is not an Integer, and the expanded macro UnitName is not a string.

Using macros in the code editors

An entirely different use of macros is to expand a macro in the code editor. To do this, press Ctrl+Space after typing the macro name - or Shift+Space, depending on your Environment Options|Editors settings. This is convenient if you create macros like: **tryf** that could expand to:

```
try
  >< // >< in macro text positions cursor after expansion.
finally
  .Free;
end;
```

Using macros in your default unit template

To demonstrate the use macros, here's the effect of using macros on the default unit which was described in "Adjusting the unit template" page 50.

```
{
<!UnitHeader!>

<!FreeWareDisclaimer!>
}

unit <!UnitName!>;

interface

// rest of the unit template is unchanged
end.
```

Notice how this template differs from the previous template:

- The company header is now placed in a macro `UnitHeader`. This saves a lot of redundant text.
- The macro `UnitHeader` contains other macros like unit name, model name, etc.
- The disclaimer is now placed in the macro `FreeWareDisclaimer`.

Diagrams

Diagrams, Diagram List view

ModelMaker supports a set of diagram types as defined by the UML. Currently the UML v1.3 style is implemented. A model can contain any number of diagrams of any type. The Diagram list view (Main menu View or F5) contains a list of all diagrams in the model. Diagrams contain symbols and associations that can be, but do not need to be linked to entities in the code model.

The Diagrams list View is used to create, rename and delete diagrams. Select a diagram in the Diagrams list view to open it in the diagram editor.

Diagrams are named. Names do not need to be unique.

Hierarchy

Diagrams can be organized hierarchically in the Diagrams view. This organization is pure visual and has no further meaning in the model. Any diagram can serve as a parent for any other diagram. To rearrange parent child relations, make sure the list’s “Order by” is set to Hierarchy (Diagram list Popup menu). Then use drag and drop to rearrange diagrams. If the Ctrl key is pressed while dropping, the hierarchy is edited; else the order within the current parent is changed.

Styles

Each diagram has a visual style and a symbol style that define the default styles for the symbol inside that diagram. Editing these styles (diagram editor popup menu Diagram attributes), affects the appearance of all symbols in that diagram. Refer to chapter “Visual styles” for more information.

Default properties

Whenever a new diagram is created (except a Cloned diagram), the format (size) and orientation (portrait or landscape) are set to the defaults as defined in the project options.

Symbols and visual containment

Symbols can visually contain other symbols. For example package symbols can contain class symbols or other package symbols; and Nodes in a deployment diagram can contain Component symbols. Visual containment implicates visual ownership only. Visual containment is not linked to the code model. Inserting a class symbol into a package that is linked to a unit will not actually move that class to that unit. It is just visualized as being part of that unit.

After a symbol is created, visual containment is fixed and cannot be changed in the diagram editor. The only way to change a symbol’s parent is to cut the symbol and paste it on the new parent. Take care: connected associations are only copied if both ends of the association are copied.

Symbol names and other visual adornments

Most symbols have a symbol name text adornment. Depending on the visual style options, the name will display a hotlink status icon, a navigation icon and the hyperlink status. Check chapters Hyperlinks and Hotlinks. The stereotype (category) of a symbol is also displayed in the name adornment.

Other visual adornments are depending on the type of symbol. For example: a state region for a concurrent state contains a state region divider and icons, a sequence diagram role symbol contains a lifeline etc.

Associations

Basics

Associations are used to model relations between symbols. Some associations such as “documentation link” and “constraint” can also be connected to other associations. An association that is not connected to both ends is invalid and will automatically be removed from the diagram.

All associations have a direction: they lead from a source symbol to a target symbol. Usually a navigation arrow is displayed is appropriate. These arrows can be suppressed in the visual style on project, diagram or association level.

Usually the visual path of an association is formed by the two association end points. Shape nodes can be inserted to create more complex paths.

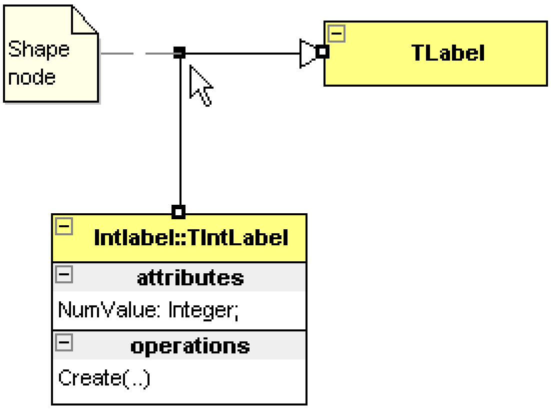
Associations are created by clicking the mouse on the source symbol and while keeping the mouse down, drag to the target symbol where the mouse is released. Once created, associations can be connected to different symbols by moving one of the endpoints to the new source or target symbol.

Anchors

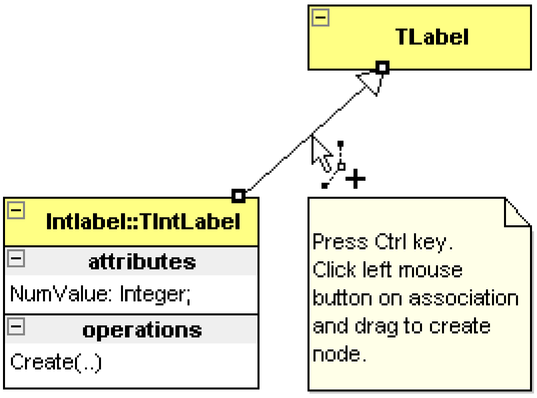
Associations are connected with an anchor point to symbols. Usually the anchor is connected to the center of a symbol. You can however drag-move the connection anchor point within the bounds of the connected symbol. This will change the intersection point of symbol and association. The diagram editor’s popup menu has a function ‘Reset Anchors” which will reset both anchors to the symbol’s center.

Shape Nodes

Shape nodes can be inserted to change the visual path of an association. Shape nodes allow bending associations visually. The association anchor points (connection points on the connected symbols) and the association’s shape nodes make up the actual visual path of the association. The picture below shows an generalization association from TIntLabel to TLabel with one shape node.



To insert a shape node into an association, press the Ctrl-key and drag on a line segment of an association. After the mouse is released, a new node is inserted. Alternatively, use Insert Shape Node from the diagram editor "Association" popup sub menu. This command is available if you invoke the popup menu on an association.

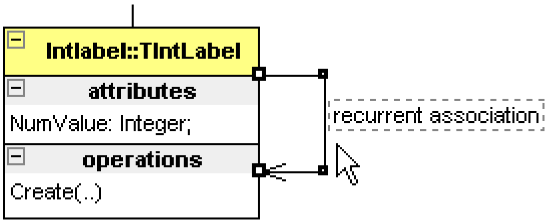


Shape nodes can be moved by selecting the association and then drag them with the mouse.

A shape node can be deleted by aligning it with the two surrounding nodes and is basically the reverse of creating a new shape node. Alternatively, use Delete Shape Node from the diagram editor "Association" popup sub menu. This command is available if you invoke the popup menu on a shape node. The command 'Clear shape nodes" in the same popup menu will clear all shape nodes in all selected associations.

Recurrent associations

Normally associations connect two different symbols. If both ends of an association are connected to the same symbol, the association is recurrent. Most associations can be made recurrent. Some recurrent associations will display a rounded curve rather than a square path. The rounded curve is a bezier that is controlled by two shape nodes. Inserting another shape node will turn the bezier curve into normal straight lines.



To create a new recurrent association, simply make the target symbol the same as the source symbol. Two shape nodes will automatically be inserted that allow control of the visual path.

An existing non-recurrent association can be made recurrent by moving on of the endpoints to the same symbol as the other endpoint. Again, two shape nodes will be inserted automatically.

To convert a recurrent association into a non-recurrent association, move either source or target endpoint to another symbol. The shape nodes will be removed.

Association Name, Qualifiers, Roles and other adornments

Most associations can be named and have an association name adornment. If the name is currently not visible, select the association and press F2 to invoke the in place editor. Depending on the visual style options, the association name will display a hotlink status icon, a navigation icon and the hyperlink status. Check chapters Hyperlinks and Hotlinks. The stereotype (category) of the association is also displayed in the name text adornment.

Other adornments include

1. Qualifiers in qualified association such as class associations. To reduce visual space, the qualifier's type can automatically be suppressed. This is controlled with the visual style options.
2. Role name, both source and target endpoint can be named in for example class associations and object links.
3. Multiplicity (cardinality), both source and target endpoint can have a multiplicity in for example class associations and actor communications.
4. Conditions, usually only in the source endpoint of sequence diagram messages and state transitions.

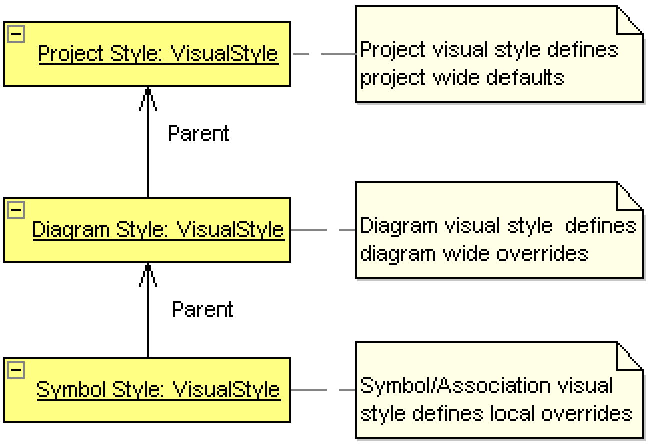
These adornments are all texts and can be moved freely in the diagram. The word break property of text adornments is controlled with the Word break functions on the "Align and Size" palette.

Visual styles

All symbols, associations and diagrams have a visual style. It is this visual style that defines how a symbol appears. A visual style contains font settings, a color palette and some options that control display of icons. The symbol styles, that are discussed in the next chapter, control *what* is displayed; the visual styles control *how* a symbol is displayed.

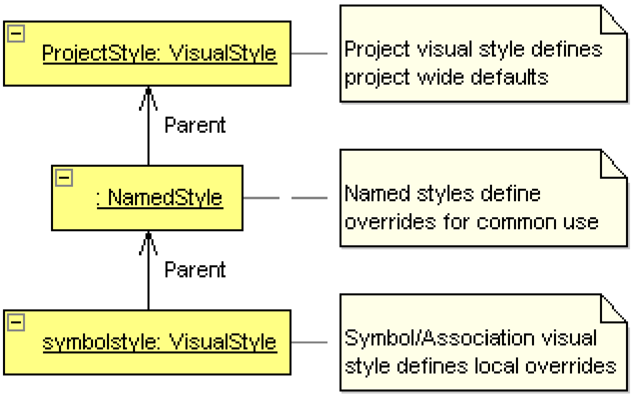
Style hierarchy

All visual styles are linked in a lookup hierarchy. A style normally looks up an attribute in the parent style but can also override "parent" attributes. By default symbol and association styles are linked to the diagram visual style and have all properties set to "lookup from parent"; that is override or change nothing. As a result, all symbols will appear as defined by the diagram visual style. On their turn, all diagram visual styles are linked to the project visual style and also have all properties set to "lookup from parent".


Project Style: VisualStyle
Project visual style defines project wide defaults
Parent
Diagram Style: VisualStyle
Diagram visual style defines diagram wide overrides
Parent
Symbol Style: VisualStyle
Symbol/Association visual style defines local overrides

This hierarchical structure allows easy adjusting of visual appearance on any level. To change the appearance of an entire project: change the project style. You can even save the project settings as default, and new projects will have the same project style. Change the diagram style to modify the appearance of all symbols a single diagram. To change the appearance of a single symbol, edit the symbol's style.

To make reuse of visual styles easy, the visual style manager allows creation of "Named styles". Named styles can be used to define a specific appearance that can be reused. A named style is applied making it the parent of a diagram or symbol style. Named styles can have other named styles or the project visual style as their parent. The diagram editor tool bar contains a parent style combo which is used for this.


ProjectStyle: VisualStyle
Project visual style defines project wide defaults
Parent
: NamedStyle
Named styles define overrides for common use
Parent
symbolstyle: VisualStyle
Symbol/Association visual style defines local overrides

Visual style properties

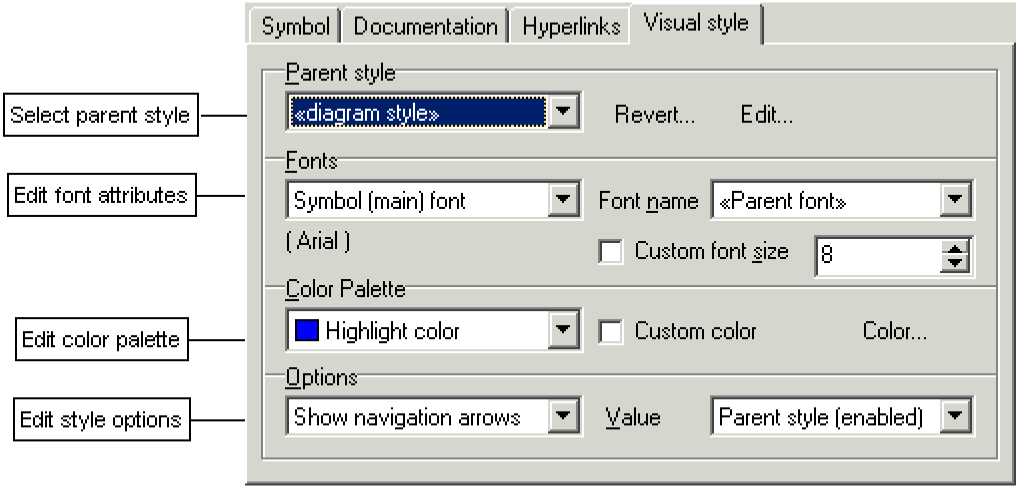
A visual style consists of a set of properties. Not all symbols / associations use all properties.

1. A main font name and size. The main font is used for symbol and association names. The font's style, bold, italic and underline, cannot be defined because that is usually has a syntactical meaning in the symbol as defined in the UML specifications: classifier names are bold, instances named are underlined etc.
2. An adornment or text font. Used for all other texts. For example the members in a class symbols and the roles in an association.
3. A color palette defining the colors for basic drawing entities such as main font, symbol compartment, symbol pen, symbol tab, association line etc. Which entries on this palette are used is dependent on the type of symbol.
4. Options that control display of some visual elements, such as: navigation arrows in associations, hotlink icons, navigation (hyperlink) icons etc.

Most symbols properties dialogs and the diagram properties dialog contain a 'visual style' tab that allows editing a style. Refer to next paragraph.

Controlling & assigning styles

Most symbols properties dialogs and the diagram properties dialog contain a 'visual style' tab that allows editing the symbol's visual style. In this tab you change and or edit the parent style and the style's attributes.


Parent style: «diagram style»

Fonts: Symbol (main) font (Arial) Font name: «Parent font»

□ Custom font size: 8

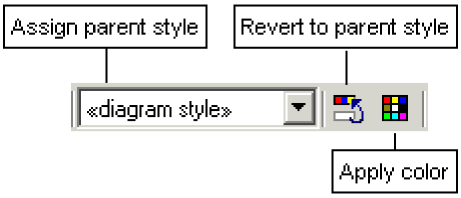
Color Palette: Highlight color

□ Custom color

Options: Show navigation arrows Value: Parent style (enabled)

To entirely revert to the parent style, erasing all overridden / redefined attributes, click the 'Revert' button.

The diagram editor's tool bar contains some visual style specific tools.

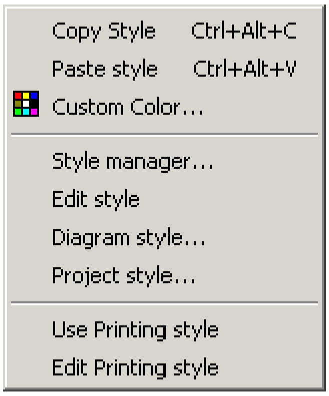


The "parent style" combo displays the parent style for the selected symbols and associations. It is blank if selected symbols have different parents. It allows assigning a new parent for the selected symbols.

The “Revert to parent style” tool will reset the visual styles for all selected symbols and associations. Useful to erase any local redefined style attributes.

The Apply color tool will let you select a color and apply it to the selected symbols and associations. The color is applied as the “main” feature color. Usually this is the symbol color palette entry, but for tabbed symbols like class symbols and package symbols, the symbol tab color is changed.

The Diagram editor contains a ‘Visual style’ sub popup menu that contains some additional visual style related functions.



Most striking functions in this popup menu are:

1. Copy / Paste a visual style. This is useful in case you have redefined the style of a symbol and want to apply the same visual style to a selection of other symbols. Note that similar functions exist for the symbol style that controls the display of members etc.
2. Show the Style Manager and it’s named styles, which is described in the next paragraph.
3. Toggle (and Edit) “Printing Style”. The printing style is described in the next paragraphs.

Style Manager

The visual Style Manager is used to create and maintain named visual styles. Named styles can be assigned as parent style for symbols and diagrams. They allow creating a predefined set of appearances that can be applied by simply assigning the style.

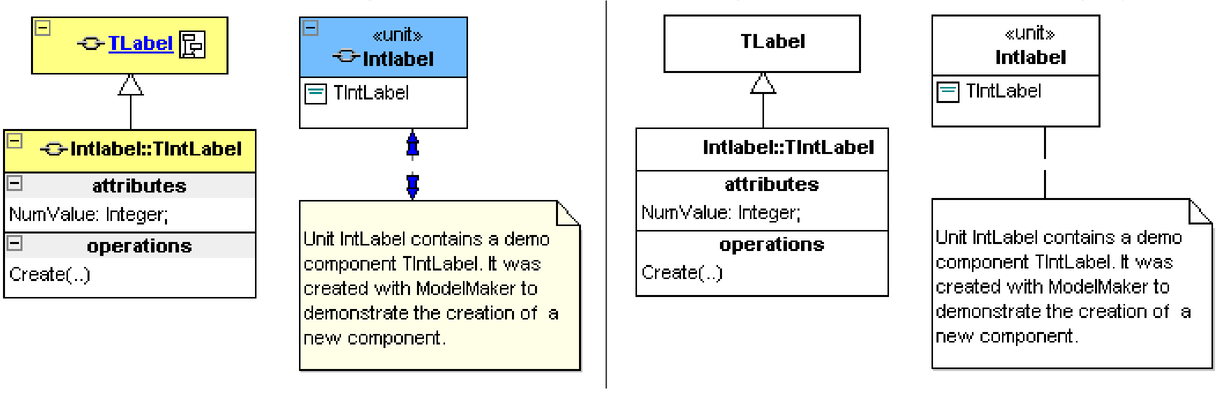
For example, you could create a style named “System components” which specifies a specific blue palette to paint symbols. Another style could be named “GUI components” and define a yellow palette to paint symbols.

In class diagrams, you can then easily change the visual appearance of a class symbol by assigning “System components” as parent style for the class symbol using the diagram editor’s tool bar parent combo.

Named Visual styles can be imported or exported with the Style Manager. This allows synchronizing named styles in projects.

Printing Style

ModelMaker allows suppressing a specific set of graphical features when printing diagrams. These include: printing in black and white (suppressing colors), no navigation icons, no hotlink icons etc. While these features can help while designing, they may be unwanted in printed output. The following picture shows the same diagram in normal and printing style.



Unit IntLabel contains a demo component TIntLabel. It was created with ModelMaker to demonstrate the creation of a new component.

Unit IntLabel contains a demo component TIntLabel. It was created with ModelMaker to demonstrate the creation of a new component.

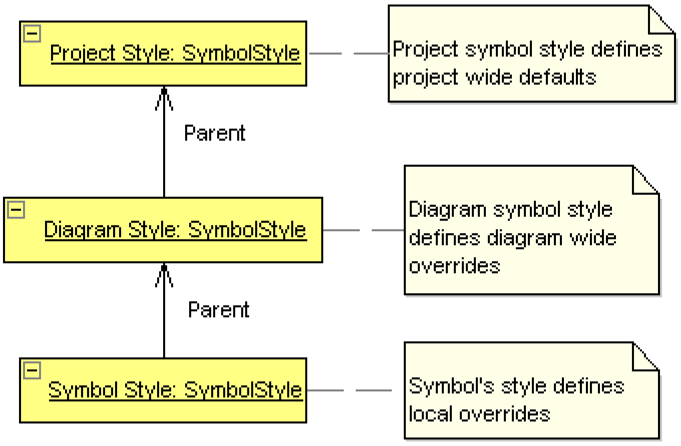
The printing style is defined in the diagram environment options. It is automatically superimposed on all other styles when printing, and can also be manually activated in the diagram editor. The diagram editor popup menu “visual style” menu contains an item that toggles the “Use Printing style” state. If the printing style is active in the diagram editor, it will also be effective when creating visual exports as image file or to the clipboard.

Symbol (contents) style

Just like visual styles hierarchically control the visual appearance of diagrams and symbols, symbol styles control the contents of displayed symbols. For example: which members are displayed in a class symbol and in which format is controlled by the class symbol “symbol style”. The symbol styles control *what* is displayed; the visual styles control *how* it is displayed. The symbol styles are only applicable for a specific set of symbols such as class, interface and (unit) package symbols. In classes the symbol style controls which members are displayed and how they are displayed. In unit packages they control whether contained classes are automatically displayed.

Style hierarchy

Just like visual styles, all symbol styles are linked in a lookup hierarchy and can override parent style attributes. Symbol styles are linked to the diagram symbol style and have all properties set to “lookup from parent” that is: override or change nothing. As a result, all symbols will appear as defined by the diagram symbol style. On their turn, all diagram symbol styles are linked to the project symbol style and also have all properties set to “lookup from parent”.


Project Style: SymbolStyle
Project symbol style defines project wide defaults
Parent
Diagram Style: SymbolStyle
Diagram symbol style defines diagram wide overrides
Parent
Symbol Style: SymbolStyle
Symbol's style defines local overrides

This hierarchical structure allows easy adjusting of what is displayed within symbols. To change the style of an entire project: change the project style. You can even save the project settings as default, and new projects will have the same project style. Change the diagram style to modify all symbols a single diagram. To change the contents of a single symbol, edit the symbol’s style.

Unlike the visual style, the symbol style cannot be linked to named styles.

Controlling & assigning styles

The diagram properties dialog contains a ‘symbol style’ tab that allows editing the symbol style. The project options “symbol style” tab is similar. In these tabs you control how class and interface symbols display members and unit packages display contained classes.

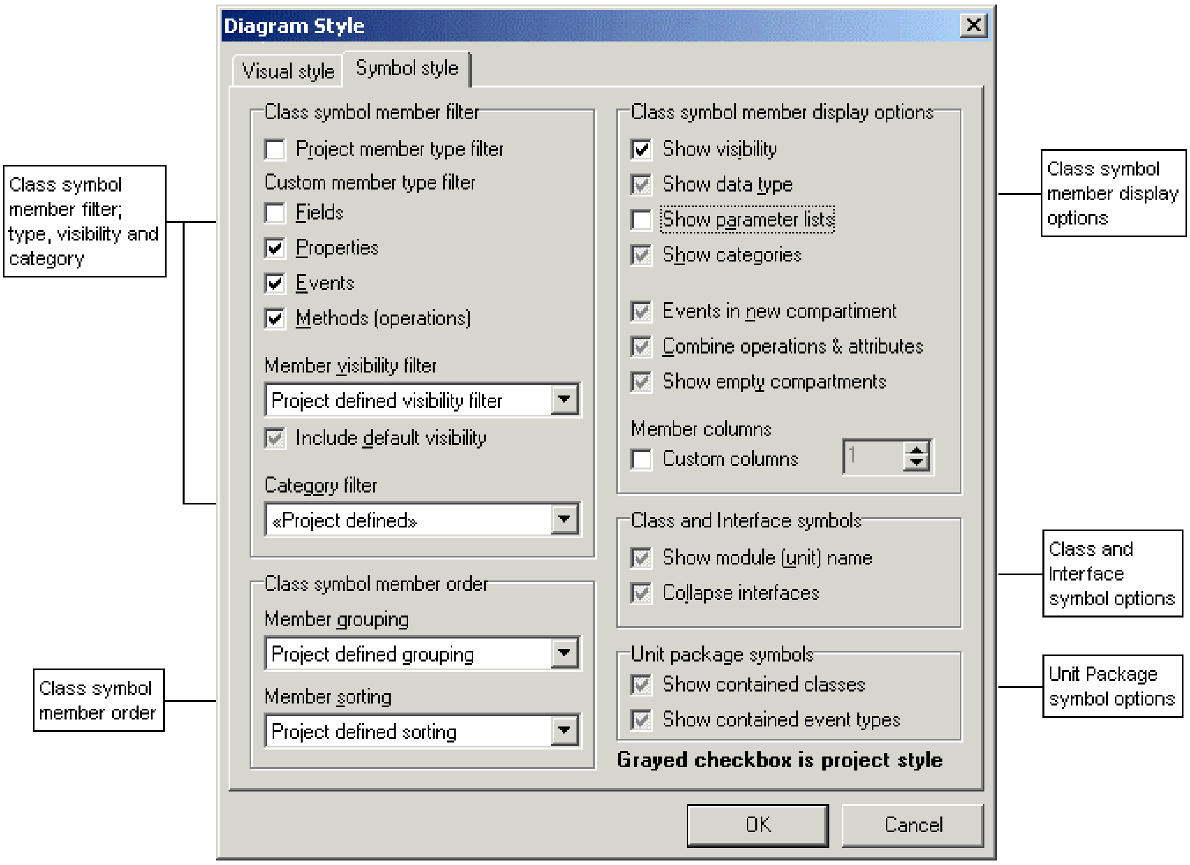

Class symbol member filter; type, visibility and category
Class symbol member order
Class symbol member display options
Class and Interface symbol options
Unit Package symbol options

Diagram Style

Visual style | Symbol style

Class symbol member filter

□ Project member type filter

Custom member type filter

□ Fields

☑ Properties

☑ Events

☑ Methods (operations)

Member visibility filter: Project defined visibility filter

☑ Include default visibility

Category filter: «Project defined»

Class symbol member order

Member grouping: Project defined grouping

Member sorting: Project defined sorting

Class symbol member display options

☑ Show visibility

☑ Show data type

□ Show parameter lists

☑ Show categories

☑ Events in new compartment

☑ Combine operations & attributes

☑ Show empty compartments

Member columns

□ Custom columns 1

Class and Interface symbols

☑ Show module (unit) name

☑ Collapse interfaces

Unit package symbols

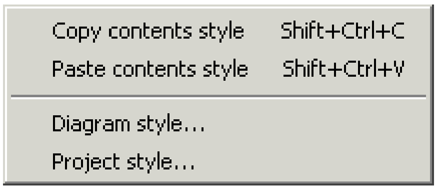
☑ Show contained classes

☑ Show contained event types

Grayed checkbox is project style

OK | Cancel

The Diagram editor contains a ‘Symbol style’ sub popup menu that contains some symbol style related functions.


Copy contents style Shift+Ctrl+C
Paste contents style Shift+Ctrl+V
Diagram style…
Project style…

Most striking functions in this popup menu are Copy / Paste a symbol style. This is useful in case you have redefined the style of a for example a class symbol and want to apply the same visual style to a selection of other class symbols.

Class & Interface symbols

The symbol style in class and interface symbols is edited on the “member style” tab of the symbol’s dialog. These are basically the same tabs as the diagram symbol style tab, except that fixed or non-appropriate elements have been removed.

Package symbols (units)

The symbol style in a package symbol is incorporated in the main symbol tab. Here you control if contained classes and events are displayed. This feature is only available for units and classes that are (imported) in the model.

Size and Alignment

The Drawing Grid

The diagram editor's drawing grid is defined in the project options diagram style tab. It helps aligning symbols. All symbols are automatically snapped to the drawing grid. And for most symbols, the extent (bounds) is automatically adjusted to fit on the grid too. The MindMap node symbol allows enabling/disabling this "Bounds on Grid".

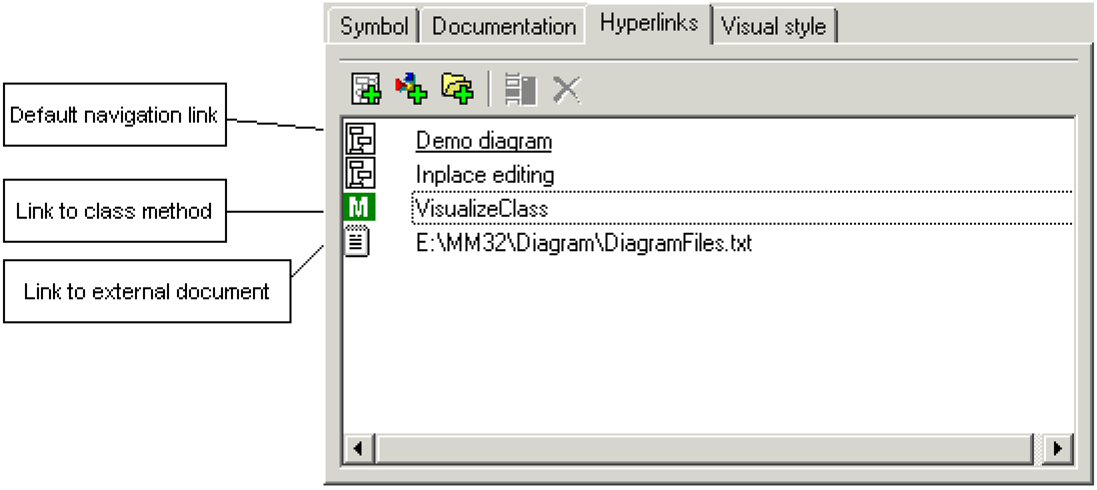
If you change the grid size, all symbols will most likely resize too. Because the grid it affects all diagrams and symbols, it is defined and saved per project.

Align & Size Palette

The Alignment palette, which is available from the diagram editor tool bar, contains a set of functions to control alignment, auto sizing, text alignment and word break properties. These functions are similar to those in other applications such as Delphi, and are not explained in detail.

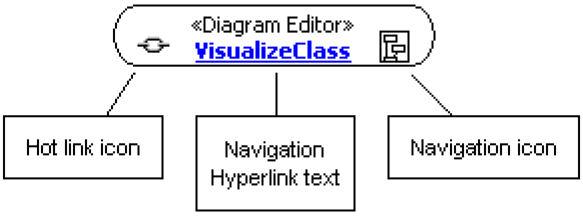
Hyperlinks, navigation

Virtually all symbols can contain hyperlinks. Hyperlinks can point to other diagrams, code model entities (class, member, event, unit) or to external documents. Hyperlinks are created and maintained in the "hyperlinks" tab of a symbol's dialog. Although there can be many hyperlinks, only one is the default navigation link. This is the first link that supports navigation. The default navigation link is underlined in the hyperlinks list. Only HotLinks to code model entities usually have navigation disabled. Check chapter HotLinks to Code Model.



Demo diagram
Inplace editing
VisualizeClass
E:\MM32\Diagram\DiagramFiles.txt

Most symbols will show a navigation icon next to their name if a navigation hyperlink is available. This is shown in the picture below. These icons can be suppressed by the visual style or the printing style. Also, if a symbol has a default navigation link, the symbol's name will appear underlined and in the hyperlink color as defined in the visual style's palette. The picture below shows this. The Hotlink icon is explained in Paragraph "Hotlinks to the code model".



If the navigation icon is clicked, ModelMaker will follow the link and navigate to the object. If this is another diagram, the current diagram is saved and the referenced diagram is opened in the diagram editor.

If the link refers to a code model entity, ModelMaker will select the entity (class, member) and make the classes and members views visible. If the entity is a method, the method editor will also be made visible.

If the link refers to an external document, ModelMaker will perform a default "open" command on the filename or URL.

Clicking the hotlink icon (left of the symbol name) will edit the hot linked entity rather than the symbol.

If a symbol contains more than one hyperlink, you can navigate to the non-default hyperlinks by using the diagram editor's popup menu Navigate function. This contains a dynamic submenu with all hyperlinks available in the focused symbol.

External documents

External documents are defined with the same alias / relative name mechanism as used for source files. Check chapter 'source aliases' in this manual for details. The use of aliases avoids the use of hard coded, machine dependent paths.

The standard shell "open" command is performed to navigate to an external document. This accepts all kinds of external documents such as executables or files that are associated with an application. For example "c:\temp\manual.doc" will be run MS word and open the document.

URLs to web pages or web sites are also valid. For example:
Alias=""
Relative filename = "http://www.modelmakertools.com"
will open a web browser and navigate to the ModelMaker Tools site. This can be used to navigate to html documentation etc.

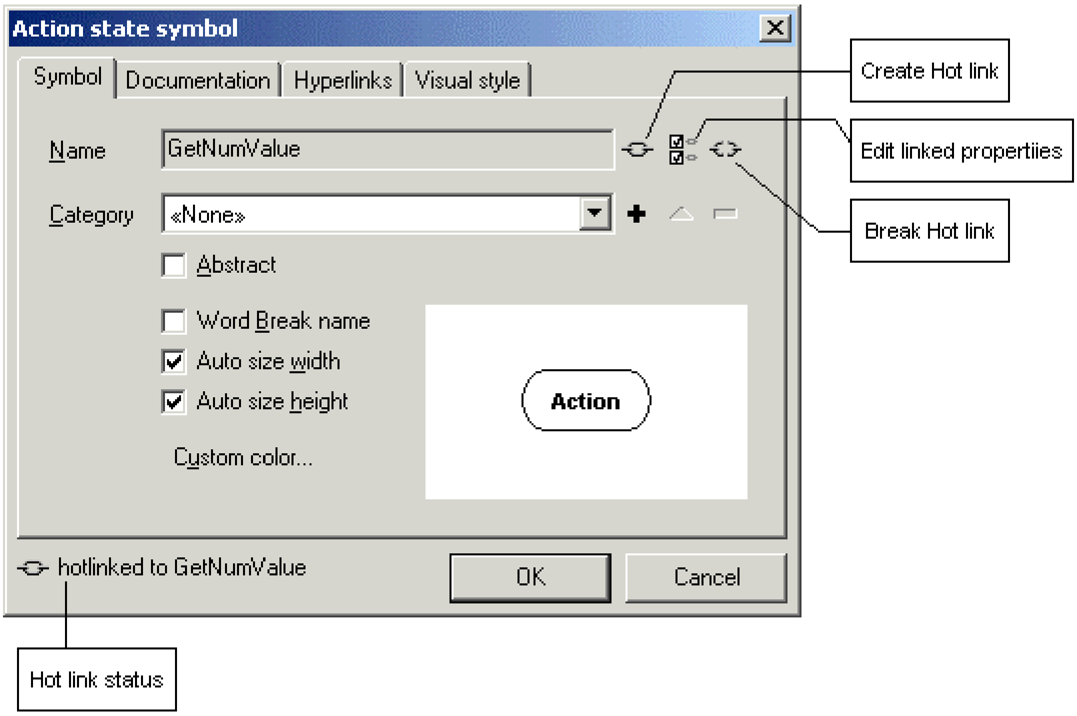
To refer to another ModelMaker model and start another ModelMaker instance, associate the ModelMaker project bundle extension *.mpb with the ModelMaker executable in the Windows shell.

Coupling Symbols to the Code Model

Most symbols and associations are linked or can be linked to entities in the code model. This is either hard coded or can be done manually by creating a HotLink. A hotlinked symbol will share name, documentation, one liner, “abstract” state and stereotype (category) with the linked entity. Modifying one of these properties in the symbol reflected to the linked entity and vice versa.

HotLinks

HotLinks are used to link a symbol or association to another entity, usually a code model entity like a class or a method. A hotlink is basically a navigation hyperlink that is (internally) marked as “hot”. Most symbol dialogs contain a set of buttons next to the name that allow creating and editing the hotlink. Here is an example from the Action State editor dialog.



Name: GetNumValue

Category: «None»

☐ Abstract

☐ Word Break name

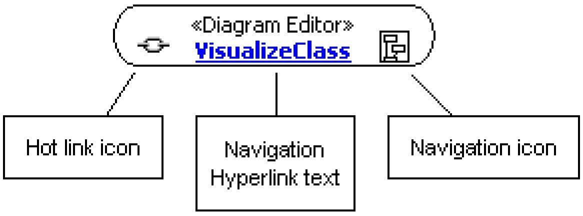
☑ Auto size width

☑ Auto size height

hotlinked to GetNumValue

Normally, action states are not coupled to any other entities. If you for example wanted to link an action state symbol to the TIntLabel.GetNumValue method, click the “Create HotLink” button and select the entity to link to. To break the hot link, click the Break hot link button. The icon at the bottom left of the dialog displays the hot link status. Not all properties need to be linked. To edit the linked properties, click “Edit linked properties”. The linked properties dialog lets you select which properties are linked.

If a symbol is hot linked, a hot link icon is displayed at the left of the symbol’s name. The following picture shows this. Hot link icons can be suppressed in the visual style and/or in the printing style.



If you click at the hot link icon in the symbol's name, the linked entity will be edited rather than the symbol.

If a symbol is hot linked to a code model entity, that entity will be selected if you click on the symbol or any of its (text) adornments. Unlike navigation through hyperlinks, this will not ensure that the associated view is made visible.

Delete HotLinked entity

The Diagram Editor toolbar contains two delete tools: one to delete the symbol from the diagram, one to delete the symbol and the hotlinked entity. The last one will not only delete the symbol from the diagram but also remove the linked entity. You will be asked for confirmation before the linked entity is deleted.

Specialized symbols and associations

Some symbols are linked to the code model by design. These symbols do not allow any other linking than the built in one.

Class and Interface symbols
Class and Interface symbols are implicitly linked to a class in the code model. If the class is deleted from the code model, all class symbols linked to that class are removed from all diagrams.

Property and Field associations
Similar to class symbols, property and field associations are hard linked to properties and fields in a class. The data type of these members must match the association target class.

Shared Class Association
Shared class associations are associations between class symbols that allow greater flexibility than field and property associations. They do not need to be linked to existing code model members or classes. Because they are shared, they can be auto visualized if both source and target classes are being visualized on a diagram.

Generalization relation

Generalization (inheritance) relations can be created between all symbols that are generalizable. In most cases they are not coupled the code model. Only if they connect two class or interface symbols, they are implicitly linked to the code model. Therefore, changing a generalization between two use cases does not affect the code model. But creating or changing a generalization between classes will be reflected in the code model by changing the class hierarchy.

Realization relation

Realization relations (such as interface support) can be created between most symbols that allow realization. In most cases they are not coupled the code model. Only if they connect a class and an interface symbol, they are implicitly linked to the code model. Therefore, changing a realization between a package and interface does not affect the code model. But creating or changing a realization between a class and interface will be reflected in the code model by adding or removing interface support to that class.

Package (units)

Units can be visualized as package symbols. Normal hot linking is used to achieve this. However, unit packages can display the contained classes and events. This is controlled by the diagram symbol style and the related options in the package editor dialog. The displayed content is read-only.

Documentation & OneLiners

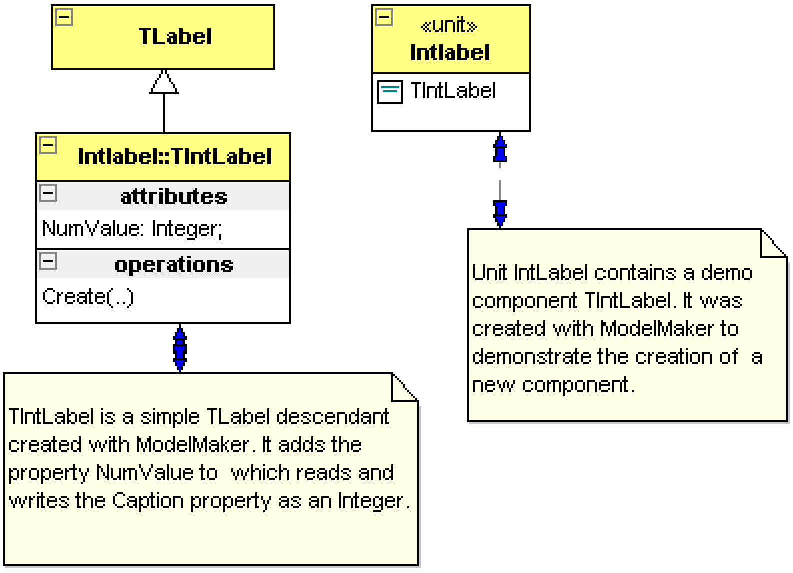
Floating Documentation view

All symbols can be documented with a One Liner and multi line documentation. Most symbol editor dialogs contain a “documentation” tab. The standard documentation view cannot be used to edit the symbol’s documentation because the diagram editor and documentation view cannot be visible at the same time. The floating documentation view however is coupled to the diagram editor’s focused symbol. This view can conveniently be used to edit symbol documentation.

If a symbol is hotlinked to a code model entity, the symbol’s documentation and one liner will be linked to that of the entity. Changing it in the entity will change it in the symbol and vice versa. This is controlled by hot links and the hot linked properties.

Linked Annotations

Annotations can visually be linked to symbols with documentation links. These links can be either passive or automatically link documentation or one liner. Here is an example from the Getting started demo in this manual.


TLabel
«unit»
Intlabel
TIntLabel
Intlabel::TIntLabel
attributes
NumValue: Integer;
operations
Create(..)
Unit IntLabel contains a demo component TIntLabel. It was created with ModelMaker to demonstrate the creation of a new component.
TIntLabel is a simple TLabel descendant created with ModelMaker. It adds the property NumValue to which reads and writes the Caption property as an Integer.

The annotations in this example are coupled in auto documentation style; the blue double arrows on the link paths show this. Changing the annotation text (for example by in place editing) will change the symbol's documentation. In the example the class symbols are implicitly linked to the code model classes. This means that editing the annotation text will modify both the symbol and class documentation. This also works the other way round: if the code model class's documentation is changed, the annotation text will be updated.

To change the style of a documentation link, double click on the link path.

Diagram Editor

Properties

The environment options "diagrams" tab controls the diagram editor's properties. These include: printing style, hint feed back, background color, grid style and color etc.

Keyboard and Mouse control

The diagram editor's keyboard short cuts are:

Scroll and Move	
Up/Down/Left/Right	Scroll
PageUp/PageDown/Home/End	Scroll one page up/down
Ctrl+PageUp/PageDown	Scroll to top/bottom
Home/End	Scroll one page left/right
Ctrl+Home/End	Scroll to left, right side of page

Zooming	
Numeric + / -	Zoom In/Out by 10%

Ctrl+Shift+I	Zoom in
Ctrl+Shift+U	Zoom out

Editing	
Escape	Cancel operation or select containing (parent) symbol
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo
F2	Rename (invoke inplace editor)
Ctrl+Up/Down/Left/Right	Move selected symbols
Ctrl+C/V/X	Copy/Paste/Cut selection
Ctrl+Alt+C/V	Copy / Paste visual style
Ctrl+Shift+C/V	Copy / Paste symbol style
Del	Delete selection
Ctrl+Del	Delete All (clear diagram)
Ctrl+A	Select All
Ctrl+P	Print Diagram
Ctrl+Alt+P	Print Preview Diagram
F12	Toggle full screen mode

Navigation	
Ctrl+U	Navigate Up; select parent diagram
Ctrl+B	Navigate Backward
Ctrl+F	Navigate Forward

Mouse selection	
Click	Select exclusive
Shift+Click	Toggle selected state, incluse in selection
Drag	Multiple Select(lasso selection)
Shift+Drag	Extend selection by lasso selection
Ctrl+Drag	Parent selection (only select symbols within parent, excluding the parent). Similar to Delphi IDE from designer

Mouse Wheel control	
Wheel	Scroll up/down
Shift+Wheel	Scroll left/right
Ctrl+Wheel	Zoom in/out

Drag & Drop and conversions

ModelMaker extensively supports drag and drop between the main model views (classes, members, units, diagrams etc). Each major model view can act as a drag source of entities and as drop target for entities dragged from most other views. The details of each combination are described in the next paragraphs. Since there are always two views cooperating in a drag /drop operation (source and target) the details of conversions are described for the target view only.

Classes view

Internal (tree mode)

- Change inheritance.
- Apply interface support, press Control to invoke interface wizard

Internal (list mode)

- Apply interface support, press Control to invoke interface wizard

Source

Acts as a source for classes (and interfaces)

- Drag a class or interface to **diagram editor** for instant visualization.
- Drag a class to a **code editor** to insert its name as text.

Target

Accepts members, local vars, procedures and event types:

- Drop **members** from the **members view** on a class to copy them to the target class. Press Shift (before releasing) to Move rather than copying the members. If a property is copied or moved, it’s read and write access members are also copied, even if not included in the dragged members. Restrictions that apply for interfaces are automatically applied: fields are ignored, property write access is restricted and visibility is made default when dropping a class’s members on an interface.
- Drop **procedures** from **Method Local Code Explorer** to convert them to new methods.
- Drop **procedures** from **Unit Code Explorer** to convert them to new methods.
- Drop l**ocal vars** from the **Method Local Code Explorer** to convert them to new fields. Press shift on dropping to move rather than copy the var. Since interfaces cannot contain fields, local vars cannot be dropped on interfaces.

- Drop **event type definitions** from **Event library** view and **Units view** (tree mode) to add an event handler or event property using the dragged event as a template. On dropping a popup menu offers the available options.

Members view

Internal

- In custom order rearrange mode (Ctrl+R toggles this mode), member custom order can be arranged.

Source

Acts as a source of members.

- Drag members the **classes view** to copy them to a class or interface.
- Drag members to a **class** in the **units view** (tree mode only). Similar to dragging members to the classes view
- Drag a field or method to the **Method Local Code Explorer** to convert it to a local var.
- Drag a method to the **Unit Code Explorer** to converted it to a local procedure.
- Drag a method to the **Method Implementation view** (toolbars, or tabs) to “pin” the method.
- Drag a method(s) to the **Event Library view** to create new event types using the methods as template.
- Drag a member to a **code editor** to insert its name as text.

Target

Accepts code sections, local vars, (local) procedures, text, event type definitions and text.

- Drop **code sections** from the **Method Implementation Section list**. The dropped section is copied to the target method. Only if dropped on a method.
- Drop a **Local var** from the **Method Local Code Explorer.** If a var is dropped on a method, the var will be copied to the target method. Press shift to move the var rather than copying it. If a var is dropped on any other member or empty space in the member list, the var is converted to a new field. Drag the vars root node in the method local code explorer instead of a single var to drag all all vars at once.
- Drop a **local procedure** from the **Method Local Code Explorer.** If a local procedure is dropped on a method, it will be copied to that method. Press shift to move rather than copy it. If a local procedure is dropped on any other member or empty space in the member list, the local procedure is converted to a new method.
- Drop a **local procedure** from the **Unit Code Explorer**. Acts similar as local procedures dragged from the Method Local Code Explorer.

- Drop **Event type definitions** from **Event Library View** and **Units view (tree mode)**. A popup menu will let you select between adding an event handler for an event type or creating an event property. Multiple event types may be dragged at once.
- Drop **Text** from the **code editors** on the “add field” “add method” “add property” and “add event” buttons in the toolbar. The corresponding member type will be created and its name will be set to the dragged text. Method parameters are extracted from the dropped text.
- Drop **Text** from **code editors** dropped on the member list acts similar as text dropped on the“add method” button.

Units view

Internal (tree mode only)

- Classes and Event types can be copied or moved between units and “Classes not assigned to any units”. Pressing Ctrl will copy rather than Move (default). Within the same unit classes and event types can be rearranged using drag and drop.

Source

In tree mode acts as source for units, classes and event types. In list mode acts as source for units only.

- Drag a unit to the **diagram editor** to visualize that unit as a package.
- Dragging a class or interface to any other view is similar to dragging a class from the classes view.
- Dragging an **event type** is similar to dragging an event type from the event library view.
- Drag a unit, class or event to a **code editor** to insert its name as text.

Target

Accepts members, members, (local) procedures, local vars (all in tree mode only) and event types (both modes).

- Drop an **event type** from the **Event Library view**. If dropped on a unit, this will add the event type definition to the unit. If dropped on a class this will add an event handler or event property using the dragged event as a template or alternatively add the event type to the unit. On dropping a popup menu offers the available options. Applies to both tree and list mode.
- Dropping entities on a class is similar to dropping on a class in the classes view (tree mode only).
- Event types contained by units do not accept dropped entities.

Method Implementation view

Method Local Code Explorer

Internal

- Rearrange local vars
- Rearrange local procedures

Source

Acts as a source for **local vars**. Dragging the vars root node will drag all vars at once. Press shift to move rather than copy / convert a local var.

- Drag a Local var (or the root "Vars" node) to a class in the **classes view** to convert it to a field.
- Dragging a Local var to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.
- Drag a local var to the **members view** to copy it to a method or add a new field in the class.
- Drag a local var to a **code editor** to insert its name as text.

Acts as a source for **(local) procedures**. Press shift to move rather than copy/convert a procedure

- Drag a local procedure to a class in the **classes view**. The procedure will be converted to a method.
- Dragging a local procedure to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.
- Drag a procedure to the **members view** to copy it to a method or add a new method.
- Drag a procedure to a **code editor** to insert its name as text.

Target

Accepts members, local vars, procedures and event types:

- Drop a **field** or **method** from **members view**. A field will be converted to a local var, a method will be converted to a local procedure. This is usually only relevant for "pinned" methods as selecting a member to drag it will automatically change the "current method".
- Drop **text** from a code editor containing an "`ident + ":" + type`" list to convert the text to local vars.

Method Implementation Section list

Internal

- Rearrange sections (drag up/down)
- Indent / unindent sections (drag left/right)

Source

Acts as a source for code sections.

- Drag a section to a method in the **members view** to copy it to that method. Press shift to move rather than copy the section.

Target

Does not act as external drop target.

Method Implementation Code Editor

Internal

- Rearrange text inside editor copy or move.

Source

Acts as a source for text.

- Drag a **text** to the members view to add a new member, using the text as name (plus parameter list for methods).
- Drag a **text** containing an "`ident + ":" + type`" list to the Local Code Explorer to convert the text to vars.

Target

Accepts all dragged entities and inserts the associated name at the drop point.

Unit Code view

Unit Code Explorer

Internal

No internal drag and drop support.

Source

Acts as a source for (local) procedures. Press shift to move rather than copy/convert a procedure.

- Drag a local procedure to a class in the **classes view** to convert it to a method.
- Dragging a local procedure to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.

- Drag a procedure to the **members view**. This is similar to dragging a local procedure from the Method Local Code Explorer.
- Drag a procedure to a **code editor** to insert its name as text.

Target

Accepts methods and procedures.

- Drop a **method** from the **members view** to convert it to a module procedure.

Unit Code Editor

Similar to Method Implementation view Code Editor.

Event Library view

Internal

Does not support internal drag and drop.

Source

Acts as a source for event type definitions.

- Drag an event to the **members view** or on a class in the **classes view**. This will add an event handler or event property.
- Drag an event to the **units view**. If dropped on a unit, this will add the event type definition to the unit. If dropped on a class in the units view (tree mode) this is similar as dropping it on a class in the classes view.

Target

Accepts methods.

- Drop a **method** from the **members view**. For each dropped method an event type will be created that takes the method's signature (name, parameter list, result type etc) as a template.

Diagrams view

Internal (tree mode only)

- In "sort hierachical" mode, diagrams can be rearranged. Pressing Shift will change diagram hierarchy relations. Other modes: no internal drag drop support.

Source

Does not act as external drag source.

Target

Does not act as external drop target.

Diagram Editor

Internal

The selected editor tool controls the extensive internal drag drop support.

Source

Does not act as external drag source.

Target

Accepts classes and units, depending on the opened diagram.

- A **class** or **interface** dragged from the **classes view** or **units view** will be visualized as class or interface symbol, object flow symbol or role symbol. For class diagrams relations with other classes are automatically visualized. If three or more classes are dropped on a class diagram the “drop visualization wizard” will be invoked. This wizard allows selecting the visualization scheme and relations to visualize.
- A **unit** dragged from the **units view** will be visualized as package symbol. Relations with classes (contained) and other packages (uses and used dependencies) are automatically visualized.

Customizing ModelMaker

Here are some links to customizing ModelMaker to your wishes. Most of them you'll find in the Environment and Project options dialogs. For details refer to the GUI reference, here are just a few:

- To adjust the appearance of ModelMaker use the Environment options
- To adjust prefixes of property access methods and fields: use Project Options|Coding style.
- To adjust the layout of the generated source code, use Project Options|Code Generation.
- To adjust the way ModelMaker imports source code: use Project options|Code Import tab.
- To adjust the in-source documentation generation and import settings, check the corresponding tabs in the project options.
- To define a basic diagram visual style: Project options|Diagram Style
- To define a basic symbol style (displayed members etc): Project options|Diagram Style
- For defining the code editor's shortcut keys, Use Environment Options|Shortcuts.

Here are some other links to customization:

- Add Version Control capabilities by using a plug in expert. (Check ModelMaker web-site for ready available third party VCS Experts)
- To adjust the unit template used for new units, refer to Adjusting the unit template, page 50.
- Create Parameterizable Code Templates for pattern like groups of members that appear in multiple classes and models.
- For creating model templates, refer to Model templates page 41.
- Define and use your own macros for use in code generation or in the code editor.
- Most views have special display settings that are controlled in their popup menu: In Members view you modify toolbar layout and sorting. In the Classes view you adjust navigation order and history etc.
- Use the MM OpenToolsApi to create you own experts.

Integration with the Delphi IDE

ModelMaker is a stand-alone application and you don't need Delphi to run it. Integrating ModelMaker with Delphi's IDE will enable some additional features.

- ModelMaker will automatically update Delphi's code editor whenever a source file is (re) generated.
- You'll be able to access Delphi's on-line context sensitive help from within ModelMaker.
- Call Delphi's 'Syntax Check' "Compile" or "Build" commands from within ModelMaker.
- Open source files and locate the member selected in ModelMaker in the Delphi IDE from within ModelMaker

Inside the Delphi IDE integration experts add several features that enable smooth integration:

- Synchronize ModelMaker with the IDE: refresh import a unit and locate the current member.
- Add (multiple) files to a ModelMaker project

Although it is possible to integrate ModelMaker with all versions of Delphi and run multiple instances or versions of Delphi at the same time, it is recommended that only one is running when working with ModelMaker, as integration is a based on a one-to-one connection.

For the same reason we suggest that you do not run multiple instances of ModelMaker. In the environment options General tab you'll find an option that will ensure this.

Integration with Delphi 3 and higher

ModelMaker is integrated with Delphi 3 and higher by use of an integration expert. These experts are automatically installed by the setup program for the IDE versions you have installed on your PC. You can manually (un)install an IDE expert later using the ModelMaker environment options "Delphi IDE" tab.

The experts add a ModelMaker main menu item to the IDE's menu bar. The ModelMaker menu contains:

- Run ModelMaker, (if not already running)
- Jump to ModelMaker (Ctl+F11 in D3, Ctrl+Shift+M in D4 or higher) - this will activate ModelMaker and select the unit, class and member corresponding to the IDE's topmost editor's position. Note that inside MM the inverse command 'Locate in Delphi' - main menu "Delphi" or main tool bar 'Locate in Delphi' - which locates the member selected in ModelMaker in the Delphi editor. ModelMaker's 'Locate in Delphi' command also has shortcut Ctl+F11.
- Add to Model, adds the topmost unit in the Delphi editor to the current MM model.
- Add files to model, lets you select which files to import in a model. In D4 and higher you may select files contained in a project or files opened in the editor.
- Refresh in Model, will re-import the topmost unit just like add to model, but only if the unit is already in the model. If the unit is not in the model, the command is silently ignored.

- Convert to Model, creates a new model and adds the topmost unit to this new model.
- Convert project to Model, which creates a new model and adds all files in the current Delphi project to the new model.

Note that the file in the IDE editor will be SAVED prior to performing the actual command. Therefore these commands won't work on read-only (project) files.

Delphi 4 and higher

The Delphi 4 and higher integration experts have additional commands in the IDE's ModelMaker menu.

- Open Model, opens the model associated with the top most file in the IDE editor. Check the web-site 'Tips' page for details.
- 'Enable Auto refresh' and 'Lock Auto refresh on Run'. These control the Auto refresh feature that is described in detail in chapter Auto Refresh Import, page 65.
- Version Control. This menu item is enabled if you integrated a Version Control system in ModelMaker using a (third party) VCS expert. The available commands depend on this VCS-expert. They usually include at a minimum Check-in and Checkout. Check the ModelMaker Tools web site for available VCS-experts.

Some additional tools and utilities

- Unit Dependency analyzer. This is the same tool as available inside ModelMaker. For a description, check the ModelMaker on-line help by pressing F1 in this tool in ModelMaker.
- Resource string wizard. This will scan a unit for hard coded strings and help in converting them to a section of resource strings or string constants.
- String to Resource string, similar to the Resource string wizard, but only handles the current string token in the editor.
- Shortcut wizard: checks the active form for duplicate keyboard hot keys in control captions like "&Apply this" and "&Surprise me" and suggests alternatives in case of conflicts.

In Delphi 4 and higher you can also add commands to the IDE toolbars. In Delphi's toolbar 'Customize...' dialog, you'll find these commands in the ModelMaker category - right click on Delphi's toolbar, go to 'all commands'.

Delphi 4 and higher syntax highlighting scheme

In ModelMaker you can specify the syntax-highlighting scheme to mimic your settings in the Delphi IDE. Unlike Delphi 1/2/3, Delphi 4 and higher do not define the default color scheme in the registry unless you manually (re-)define it. If the syntax highlighting scheme in ModelMaker is set to "Delphi 4" or higher, it might display strange settings: anything could happen such as underlined, blue colored normal text.

In order to solve this problem, in Delphi 4+ go to environment options and on the Colors tab define all fore-and background colors you want by explicitly selecting them rather than relying on the 'use default' check boxes. The same applies for the font styles: you must explicitly select them. After applying these settings and restarting ModelMaker, you should have the highlighting scheme you selected. The following entries should be explicitly defined: "Comment"," Identifier", "Number", "Plain text", "Reserved word", "String", "Symbol", "White space" and "Marked block".

Uninstalling IDE integration experts

If after uninstalling ModelMaker you still get a message when starting Delphi 3 or higher saying: can't find ..\\..MMEXPT.DLL or similar, you must manually uninstall the ModelMaker integration experts.

To do this:
Either use the ModelMaker environment options 'Delphi IDE' tab to uncheck the IDE version you want uninstall, or
Run RegEdit.exe from the "Start" menu and go to
HKEY_CURRENT_USER\Software\Borland\Delphi\3.0\Experts
There should be an entry called ModelMakerExpert, to uninstall the expert you must manually remove that entry. Higher versions of Delphi have a similar registry entry

Integration with Delphi 1 and 2

Integration with Delphi 1 and 2 does not offer the same functions as the Delphi 3 and higher integration experts. Only basic synchronization functions are supported. And the installer cannot activate the integration - you must install the integration yourself. Integration consists of two aspects:

1. Installing the unit MMINTF.PAS (in directory [installdir]\mmintf in your component library. Installing this unit will enable ModelMaker to:
    - Automatically update Delphi's code editor whenever a source file is (re)generated.
    - Access Delphi's on-line context sensitive help.
    - Call Delphi's 'Syntax Check' command from within ModelMaker.
2. Installing the utility UNITJMP.EXE in your Delphi Tools menu. This will enable you to jump from Delphi's code editor to the corresponding code in ModelMaker and perform some basic file related commands.

Installing the integration unit in Delphi 1 /2

1. Start Delphi 1 /2
2. Add the unit `ModelMaker\6.0\Mmintf\MMIntf.pas` to your component library, just like you would do with any other component (refer to your Delphi user manual).
   In Delphi 1: Select menu 'Options|Install Components',
   In Delphi 2: Select menu 'Component|Install',
   select Add and browse to find the unit `MMINTF.PAS` in folder `ModelMaker\6.0\MMintf\`.

3. Don’t be surprised that you won’t see any changes in your VCL component palette after Delphi recompiled the VCL: there is no new component installed, just an integration link, which is not a component.

Installing UNITJMP.EXE as a DELPHI 1 /2 IDE tool

Installation of UNITJMP is basically the same for Delphi 1 and 2. To install the UNITJMP.EXE utility, (refer to your Delphi user manual)

1. Start Delphi
2. Select menu ‘Options|Tools...’. (*Delphi1.0*)
   Select ‘Add’ and add a new tool, title it ‘Jump &to MM’. (D*elphi 1.0*)
   Select ‘Add’ and add a new tool, title it ‘Jump to &MM’. (D*elphi 2.0*
3. Select ‘Browse’ to locate the UNITJMP.EXE file in the `ModelMaker\6.0\BIN` folder.
4. Select ‘Macros’ to pass the parameters ‘`$ROW $EDNAME`’, the space is required; the single quotes (‘’) should not be entered.
5. Select OK and Close to add this utility.

Now you’ll be able to jump from Delphi’s code editor to ModelMaker (which should be running) by pressing `Alt+T+T` (D1) or `Alt+T+M (D2)`

UnitJump can be installed more than once as a tool to perform different integration tasks, each time passing different parameters.

1. For automatic refreshing of the top most file in the IDE editor, pass parameters '-1 $EDNAME', refer to “Refresh Import” for details. You could enter ‘&Refresh Import” as title.
2. To add the topmost in the Delphi editor to the current model, use parameters '-3 $EDNAME'.
3. To create a new model and add the topmost in the Delphi editor to this model, use parameters '-2 $EDNAME'. You could enter ‘&Convert to Model’ as title.

Yes: you have identical “tools” `UNITJMP` now; the only difference is the parameters passed.

MMToolsApi primer

This chapter is a introduction on COM interfaces and using the MMToolsApi to create your own experts to ModelMaker's functionality. You should also check the MMExptDemo.dpr that demonstrates most aspects of an expert. If you have not noticed yet: in the ..\ModelMaker\demos directory there's a model MMToolsApi.mpb that contains two diagrams showing relations of the API. Do not use this model to re-create the MMToolsApi.pas, this file will be changed in future versions.

Interfaces basics

Interfaces are like classes in the way that once you've got a pointer to an interface, you can call methods and read/write properties.

Assume for example you have an interface pointer CodeModel: IMMCodeModel. From the MMToolsApi unit you can see that this interface supports the ClassCount and Classes[idx] properties. Also you can see that Classes[idx] returns IMMClass interface pointers. You could now for example iterate the code model for classes and list their names in a list box

```
for I := 0 to CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

Expert DLL basics

The big thing is now: how do you get the first interface pointer CodeModel, because once you've got hold of that, everything is easy. In the MMToolsApi there is a central access point called MMToolServices: IMMToolServices that is declared as a global var in unit MMToolsApi. This interface var is initially nil but gives access to all major aspects of the MM engine such as the CodeModel in the above example. ModelMaker Experts are dll's which are loaded dynamically (all experts must be placed in directory ..\[installdir]\experts.) After loading the library MM looks for a procedure called MMExpertEntryProc, as defined in MMToolsApi.pas and calls this procedure passing the interface pointer of the actual MMToolServices object. You should store this interface pointer as it provides your central access the tools API. You can access this interface and read the CodeModel: IMMCodeModel interface as shown in this example

```
procedure EntryProc(const Srv: IMMToolsServices); stdcall;
begin
  // here you get passed the interface pointer, store it to use later.
  MMToolServices := Srv;
end;
```

The entry procedure should be exported named SMMExpertEntryProc (MMToolsApi.pas). There's also an exit procedure, which is called upon termination of ModelMaker. In this procedure we use the previously stored MMToolServices as central access point:

```
for I := 0 to MMToolServices.CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

MMToolsApi version control

The unit MMToolsApi is continuously under construction and version control is governed by an ExpertVersionProc. This is the first procedure ModelMaker attempts to call when loading the expert. In your expert you must export this function and return the MMToolsApiVersion constant as defined in unit MMToolsApi.pas. ModelMaker will reject any expert not exporting this procedure or experts that do not match the version of the MMToolsApi with which ModelMaker was created.

```
function ExpertVersion: LongInt; stdcall;
begin
  Result := MMToolsApiVersion;
end;

exports
  ExpertVersion name SMMExpertVersionProc;
```

Interfaces and memory management

Memory management for interfaced classes is controlled by reference counting which is automatic done for you by the compiler. Even assigning an interface to a local var will increase the reference count automatically, as will assigning nil to the var or going out of scope (exiting a procedure) will decrease the reference-count again. This means you don't have to worry about freeing classes after creating them. In general all interfaces passed on from ModelMaker to you are objects existing in MM.exe space. All interface pointers you pass on to ModelMaker are objects you create (such as an expert) and ModelMaker will only have access through the interface pointer. Since these objects are reference counted the objects will disappear after MM and the expert both drop all references to it.

For example if you retrieve an IMMClass interface pointer from ModelMaker and use it in your expert code, the actual interface object will exist as long as you keep a reference to it for example by assigning to a global var. The actual class the IMMClass refers to, may be gone in ModelMaker due to user actions (delete the class, open a new model) but the interface object remains live until the expert drops all references. To check whether an interface is actually connected to a real class you could /should check the Valid property. If Valid is False, the actual class object does not exist anymore, just the interface object. The interface object will return default values in all functions (such as GetName =”).

Adding an expert and menu items

After reading the COM interface basics, you might want to create something useful which initiates on user action. The MMToolsApi provides the IMMExpert mechanism for this. You can create an object implementing IMMExpert and register it in ModelMaker, again using the MMToolServices. The fun about IMMExpert is that it allows you to insert menu-items in the ModelMaker main menu bar Tools menu and some predefined pop-up menus. Each expert should support properties Verbs, VerbCount and MenuPositions. VerbCount defines the number of menu-items you want inserted and Verbs are the actual menu item Captions. MenuPositions[..] defines where to which menu an item should be added. If the user clicks one of these menu-items, the Execute(Idx) method of the expert will be called, where Idx is [0..VerbCount -1]

There are some more methods supported by IMMExpert, but these are the basics. In the MMExptDemo.dpr you'll find an object implementing IUnknown and IMMExpert. If you go through the associated code you'll see that it has only one Verb and does only one thing in Execute. This demo expert could serve very well as a base for all other experts.

How do you inform ModelMaker that you want such an expert to be installed? Well create it (instantiate the class) and register the interface pointer with ModelMaker using MMToolServices.AddExpert(..). Reference counting will again take care of memory management. On shutdown you should remove your expert again using the index that was passed by AddExpert.

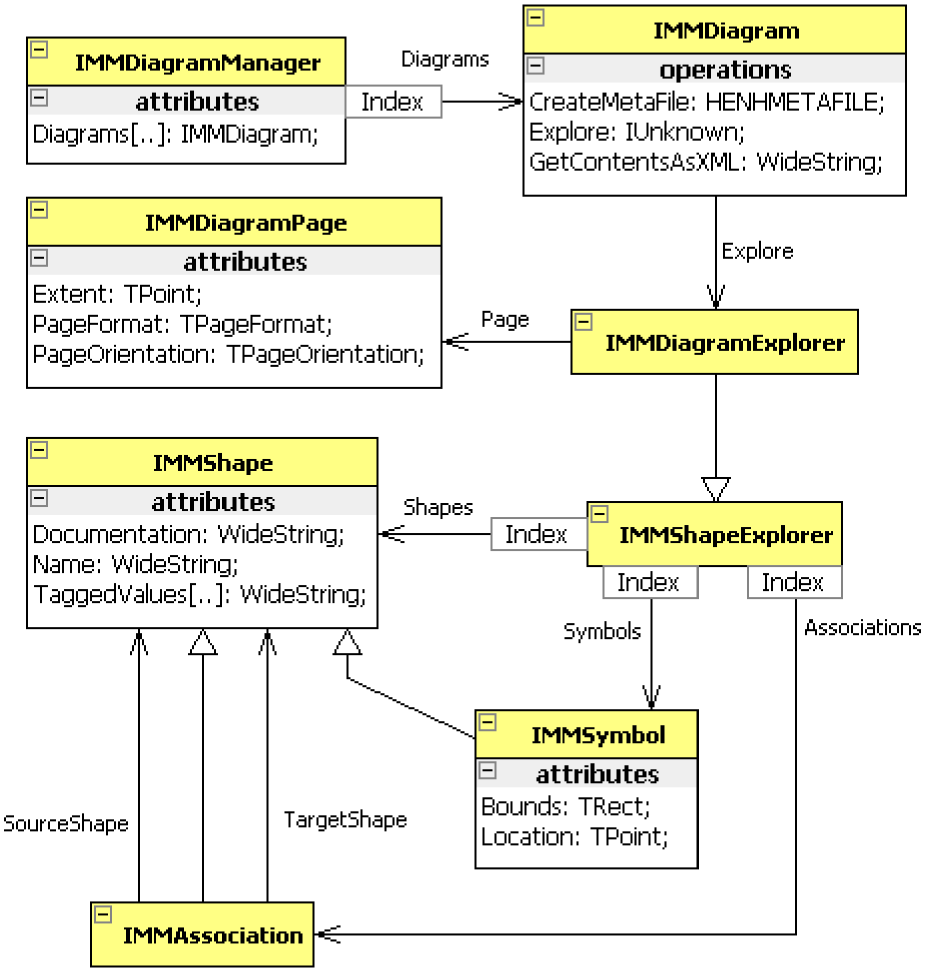
```
var
  ExptIndex: Integer = -1;

procedure MMExpertEntryProc(Srv: IMMToolServices); stdcall;
begin
  MMToolServices := Srv;
  ExptIndex := MMToolServices.AddExpert(TMyExpert.Create);
end;

procedure MMExpertExitProc; stdcall;
begin
  if ExptIndex <> - 1 then MMToolServices.RemoveExpert(ExptIndex);
end;
```

Suppose you want to create an expert that does two reports "Simple" and "Extended", you should create an object that returns VerbCount = 2 and Verb[0] = 'Simple', Verb[1] = 'Extended'. To add the items to the Tools menu, return mpToolsMenu in GetMenuPositions. Then in Execute(Idx) you check whether Idx = 0 or 1 to either create the Simple (0) or extended (1) report. The actual report creation could be something like:

- Create a form containing a list box
- use
  ```
      for I := 0 to MMToolServices.CodeModel.ClassCount do
        ListBox.Items.Add(CodeModel.Classes[I].Name);
  ```
  to fill the list box, and
- Call the Form's ShowModal method to show the current class list

Accessing Diagrams through the API

The following picture explains how to access the Diagrams and their contents through the MM ToolsAPI.



The IMMDiagramManager property of the MMToolsServices is the entry point that gives access to all diagrams. The IMMDiagram interfaces are life pointers to the actual diagrams as displayed in the diagram list view. To get at the contents of a diagram, use the IMMDiagram.Explore method. This will create a Diagram Explorer similar to the diagram editor. The interface this method returns can be cast as an IMMDiagramExplorer. The definition of IMMDiagramExplorer and the symbols can be found in MMDiagramAPI.pas.

Note that each time you call Explore, as new explorer is created. And that the contents of two explorers are not linked. If you create two explorers and modify a diagram in explorer_1, in explorer_2 you won’t see the change made in explorer_1.

An explorer gives access to a diagram’s symbols and associations. The Shapes property simply concatenates the symbols and associations properties. An IMMShape gives access to basic shape behavior: name, documentation and hyperlinks. An IMMymbol gives acsess to symbol specific properties like Bounds and Location (which associations do not have). IMMAssociation defines the association specific properties like SourceShape and TargetShape.

There are many ways to manipulate a diagram, the ModelMaker Tools demo expert shows a few examples like: create a sequence diagram and create an image containing a single class.

Accessing Experts through scripting

ModelMaker 6 is a self-registering COM server that allows access to plug-in experts that support IDispatch. The ModelMaker type library can be found in the [installdir]\experts directory. It contains interface IApp that contains a single method: GetExpert.

```
  IApp = interface(IDispatch)
    ['{D077CEC1-83F0-11D5-A1D2-00C0DFE529B9}']
    function  GetExpert(const ExpertID: WideString): IDispatch; safecall;
  end;
```

The parameter ExpertID is used to locate the expert based on the value returned by IMMExpert.ExpertID.

If your expert supports IDispatch and inherits from TAutoObject, you can access it for example in a java script like this (assuming your expert has a method named TestMethod which takes a single WideString parameter).

```
// Java script
var mm = new ActiveXObject("ModelMaker.App");
var api = mm.GetExpert("ModelMakerTools.ScriptingDemoExpert_10");
api.TestMethod("Hello World");
```

The above example will start ModelMaker or locate the active instance. Then locate the test expert and call it’s method named “TestMethod”.

This mechanism can be used to expose specific interfaces to scripting. For example, assume you have a reporter plug-in that supports this interface:

```
type
  IMyReporter = interface(IDispatch)
    procedure CreateReport(const ModelName, ReportName: WideString); safecall;
end;

  IMyReporterDisp = dispinterface
    ['{F9BA1301-84EB-11D5-A1D2-00C0DFE529B9}']
    procedure CreateReport(const ModelName, ReportName: WideString); dispid 1;
  end;
```

You could then call this expert from a script to load a model and create a report.
To learn more about disp interfaces and IDispatch, please check the Borland Delphi developers guide.

THOUGHT SMITHY

Getting Started with ModelMaker

Introduction

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training

ModelMaker is an extremely powerful tool; as a result, like all powerful and extensive products, ModelMaker is complex and can be quite daunting to a new user. With that in mind, I've prepared this document as a starting place for those who see the potential in ModelMaker and wish to get up and running faster.

ModelMaker is often billed as a UML diagramming tool or a Delphi CASE tool. However, it is far more than a diagramming tool and the label "CASE tool" sometimes brings to mind some AI attempting to write code for you. A more accurate description of ModelMaker is that of a full-featured, extensible two-way code management tool with support for UML diagramming, design patterns, reverse engineering, refactoring, etc.

At its core, ModelMaker features an "active code model" — all your classes and associated elements (units, diagrams, documentation, event types, etc.) are objects within the model. ModelMaker provides various views into this active model, allowing you to manipulate it from within a class list, member list or a diagram… When you're ready, you can generate source code units from the model to be compiled by Delphi. Since the units are generated afresh each time (rather than MM working in the extant units), you can change various settings (such as code commenting options, code order, method instrumentation, etc.) and regenerate the units for a variety of purposes (instrumented code for debugging, heavily commented code for automated documentation generation, etc.).

Note that these tutorials are in no way comprehensive; there is more than one way to do most operations; and there are many, many options in the environment. But the following topics should give you a small taste of what ModelMaker is capable of. For more information or assistance, visit www.modelmakertools.com for downloadable documents. You can also request ModelMaker training services via Thoughtsmithy; see our Training Services page for details or contact us at info@thoughtsmithy.com.

THOUGHT SMITHY

Getting Started with ModelMaker

Tutorial 1 — Importing (and Generating) Code

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training

Warning: *in order to avoid overwriting some of your production code, make a copy of the unit or units you are going to import before continuing.*

The fastest way to see ModelMaker in action is to import some of your existing code.

Open ModelMaker and create a new model by selecting the New button or File|New.

At this point we are going to tell ModelMaker to import a source file — it will reverse engineer the code it finds and add it to the model. Keep in mind that while ModelMaker's parsing engine is good, it can only deal with so much obfuscated code formatting before it chokes. For most people, this is not an issue — if MM encounters problems while importing your code, it will tell you.

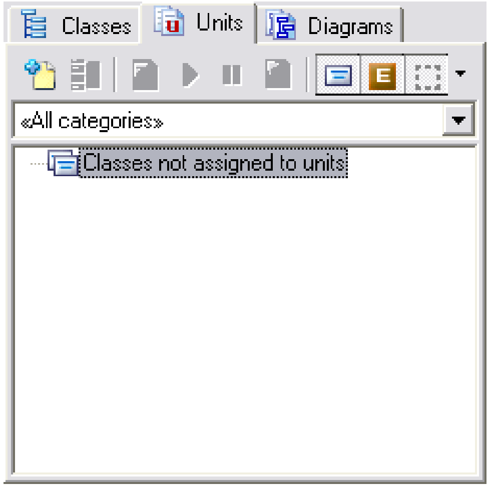


Figure 1 — the empty Units View.

There are multiple ways to import a file, but the quickest way is via drag & drop.

In ModelMaker, select the Units View by pressing F4 or by selecting its tab.

Find the unit you wish to import in Windows Explorer and then position ModelMaker so that you can drag the file into MM's main window.

Drop the unit into the Units View — if you're in a new project that is empty, the Units View should look like figure 1.

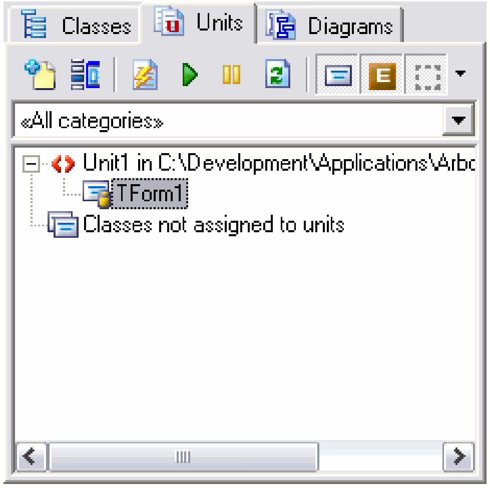

Classes
Units
Diagrams
«All categories»
Unit1 in C:\Development\Applications\Arbc
TForm1
Classes not assigned to units

Figure 2 — Units View with imported unit.

Once you have dropped the unit, it will be reverse-engineered. The unit and any classes it contains will be added to the Unit List, which should now look something like Figure 2.

Don’t worry if your icons look a little different — I’ve got some ModelMaker extensions installed that can change the appearance of my screens.

The top-level node in the Unit View treeview represents the unit you’ve just imported. Any child nodes are the classes, interfaces or event types that reside in that unit. You can double-click these nodes to bring up their editors.

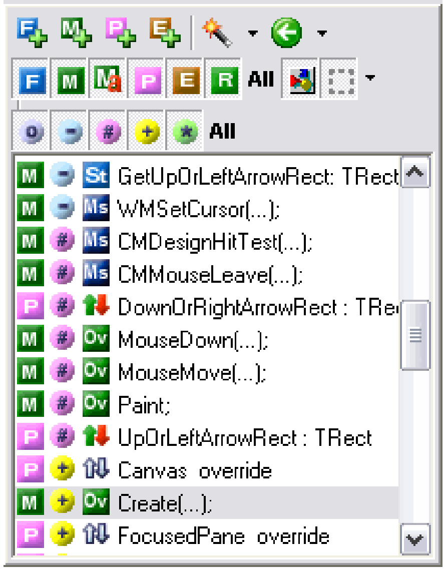


Figure 3 — the Member List.

Note that if you select a class in the Unit View, all of its members (properties, methods, etc.) will be displayed in the Member List, which looks something like figure 3.

We’ll cover the Member List in Tutorial 5; for now it’s enough to notice that your class’ members are enumerated here.

Press F3 to switch to the Classes View. You should notice here a hierarchical representation of the classes you’ve imported. Figure 4 is an example.

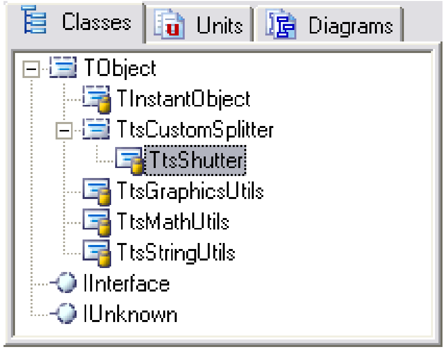


Figure 4 — the Classes View.

Notice that TObject, IInterface and IUnknown are always present. Note also the appearance of my TtsCustomSplitter class in Figure 4 — this is the ancestor of TtsShutter, which I’ve imported. However because I did not (yet) import the unit containing TtsCustomSplitter, this class must be represented as a “placeholder” (note the dotted lines around the class icon which denote placeholder status.) ModelMaker knows that TtsShutter descends from TtsCustomSplitter, but that’s all it knows. If I wish to be able to use ModelMaker’s features involving inheritance, I’d have to import TtsCustomSplitter’s unit.

ModelMaker is also capable of importing your in-source comments and attaching them to their associated entities, but only if the comments and/or ModelMaker are properly set up to do so. If MM’s Unit Code Editor (F7) shows a mass of unassociated code comments, fear not — we’ll cover importing comments in Tutorial 7.

This is an extremely abbreviated example of how to import existing code. You can also perform imports using one of the two import buttons on the main toolbar.

– this button will import your code into a new model.

– this button will import your code into the existing model.

In both cases, the buttons will give you the option of importing from a specific path or from a Source Path Alias (which we have not covered here). See Source Path Aliases in the ModelMaker help file for more information.

Generating Code

Obviously our ultimate goal is to create code which can be compiled in Delphi. ModelMaker uses its internal code model and applies the various formatting options you've selected to generate the specified units.

— the *Unlock Code Generation* button.

— the *Lock Code Generation* button.

— the *Generate* button.

— the *Enable Auto Generation* button.

— the *Disable Auto Generation* button.

In its default configuration, ModelMaker will not generate source code until you tell it to. To do so now, switch to the Units View by pressing F4 or by selecting the Units tab. Assuming you have one or more units containing classes (if you don't, go back, use your new-found skills and import some units), first, make sure that code generation is enabled by clicking the Unlock Code Generation button in the main toolbar (see sidebar), then select a unit to generate and press the Generate button (found in the Units View toolbar — see sidebar.) You should find that the specified unit has been regenerated by ModelMaker. If the unit was open in Delphi, you'll see that Delphi has already reloaded the new version of the file. This method is handy if you don't want to generate the code until you are ready.

However, if you wish, ModelMaker can be set to automatically regenerate a unit every time you make a change to that unit. This is called Auto Generation. You can enable or disable Auto Generation for each unit in your model by clicking either the Enable or Disable Auto Generation buttons in the Units View toolbar (see sidebar). Now, anytime you make a change to an Auto Generated unit, ModelMaker will regenerate the file. For that reason, care should be taken when enabling Auto Generation.

There are several other options and Delphi integration features which can effect when and how code is generated; we'll take a look at these options in Tutorial 6.

Return to the Introduction.
Go on to the next tutorial.

Getting Started with ModelMaker

Tutorial 2 — Visualizing Imported Classes

by Robert Leahey of Thoughtsmithy

To get an initial taste of ModelMaker‘s diagramming capabilities, let‘s visualize one of the classes we imported in the last tutorial.

ModelMaker‘s split screen design allows you to choose from three different possible views on the left of the screen and eight on the right. The views on the left (“master views” I‘ll call them) are:

Classes View (F3)
Units View (F4)
Diagrams View (F5)

The views on the right (“detail views” or editors) are:

Method Implementation Editor (F6)
Unit Code Editor (F7)
Diagram Editor (F8)
Macro Editor (Shift + F6)
Design Patterns (Shift + F7)
Differencing (Shift + F5)
Documentation (Shift + F8)
Event Types (Ctrl + F8)

The various views can, of course, be accessed via their tabs, so you don‘t have to remember the keyboard shortcuts.

For our demonstration, first we‘ll want the Diagrams View, so press F5.

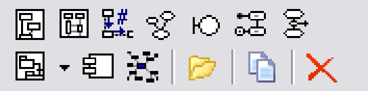


Figure 5 — the Diagram View toolbar.

From the Diagrams View toolbar, (figure 5) click the Add Class Diagram button. Now that we have a new class diagram open, we‘ll want to see the Classes View on the left while editing the diagram on the right, so press F3. Depending on how ModelMaker‘s environment options are set, sometimes changing the master view can change the active editor, so if your diagram goes away, press F8 to bring it back.

There are several ways to add a class to a diagram, but again, the easiest way is to drag it from the Classes View and drop it into the diagram. So select a class from the Classes View, click-and-drag it into the class diagram and drop it there. With the default settings, ModelMaker will display only the class name in the symbol, like in figure 6.

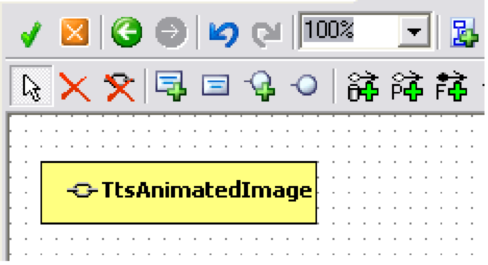
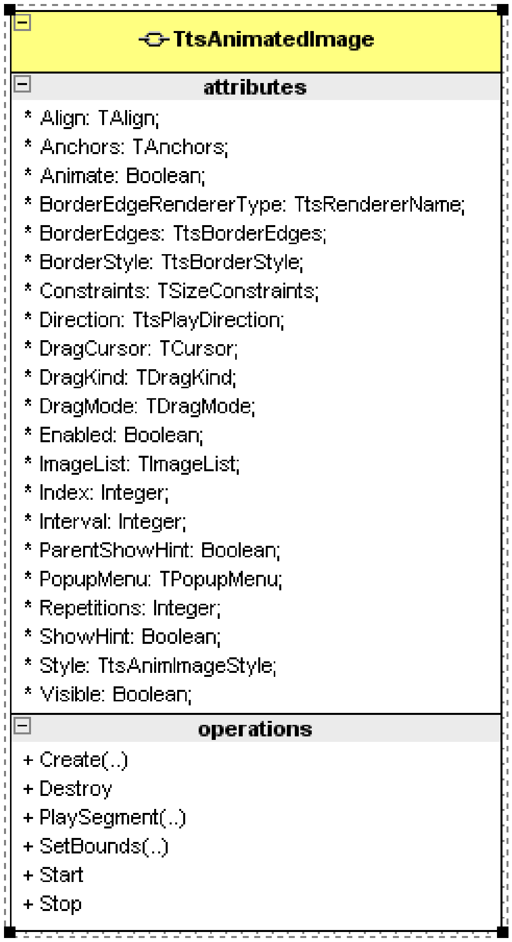


Figure 6 — a visualized class.

To cause the diagram to display some or all of the members of your class, we‘ll need to change the display style properties. ModelMaker‘s diagram display styles use an inheritance scheme. You can set properties for a symbol, but by default it will inherit the display properties of the diagram. The diagram will inherit a project‘s styles unless you override them, and likewise, a project will inherit the environment‘s styles.



TtsAnimatedImage

attributes

* Align: TAlign;
* Anchors: TAnchors;
* Animate: Boolean;
* BorderEdgeRendererType: TtsRendererName;
* BorderEdges: TtsBorderEdges;
* BorderStyle: TtsBorderStyle;
* Constraints: TSizeConstraints;
* Direction: TtsPlayDirection;
* DragCursor: TCursor;
* DragKind: TDragKind;
* DragMode: TDragMode;
* Enabled: Boolean;
* ImageList: TImageList;
* Index: Integer;
* Interval: Integer;
* ParentShowHint: Boolean;
* PopupMenu: TPopupMenu;
* Repetitions: Integer;
* ShowHint: Boolean;
* Style: TtsAnimImageStyle;
* Visible: Boolean;

operations

+ Create(..)
+ Destroy
+ PlaySegment(..)
+ SetBounds(..)
+ Start
+ Stop

Figure 7 — the class with attributes and operations.

In the interest of time, let‘s adjust the symbol display properties for the whole diagram. Double-click the diagram anywhere outside your class‘ symbol. This will bring up the Diagram Style dialog. Select the Symbol Style tab and uncheck the Project Member Type Filter checkbox. We‘re telling ModelMaker that we want to override the project‘s display style for this diagram. Now that the checkboxes under Custom Member Type Filter are enabled, check the Properties and Methods checkboxes. We‘ve told the diagram we want it to display any class symbol‘s properties and methods. Click OK.

Your class symbol should now resemble something like figure 7 — a class with attributes and operations displayed. You should note that the class symbol has some active hot spots — the tiny minus sign in the upper left hand corner of each section will collapse that section (these appear when the class is selected); the little linked chain icon next to the class name will open the class editor if clicked.

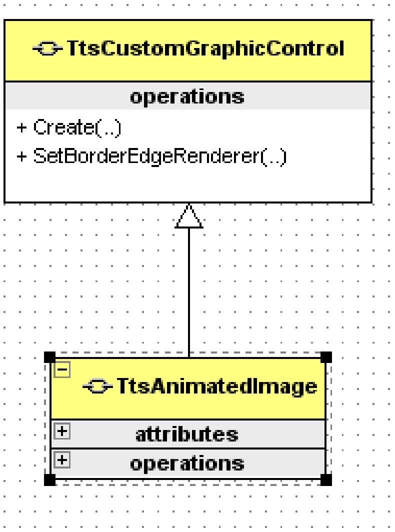


Figure 8 — automatically visualized generalization relation.

If your model contains an ancestor for the class you‘ve just visualized, drag the ancestor to the diagram and drop it there. Notice that ModelMaker automatically visualizes the generalization relation between the two classes as in figure 8.

This topic describes how to visualize an existing class from the code model. It is also possible to add a new class to the code model by adding it to the diagram. For more information, read the next tutorial.

Return to the Introduction.
Go back to the previous tutorial.
Go on to the next tutorial.

THOUGHT SMITHY

home | software development | training services | print | audio productions | lego sculptures | contact us

Getting Started with ModelMaker

Tutorial 3 — Creating Classes Within Diagrams

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training

Now that we‘ve visualized existing classes, we can examine how to create a new class within a diagram. As we mentioned earlier, the master views in ModelMaker (Classes View, Units View and Diagrams View) are all just different views into the same active model. This means that we can add a model element in any of these views — adding those elements is just a slightly different process in each view.

For our purposes, you can either use the class diagram from the previous topic or create a new one. As a point of convention, it‘s important to note the differences between a few of the tool buttons on the Class Diagram Editor toolbar (see figure 9). There are some buttons here that are common to all diagrams and we‘ll discuss these in the next tutorial.

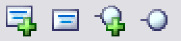
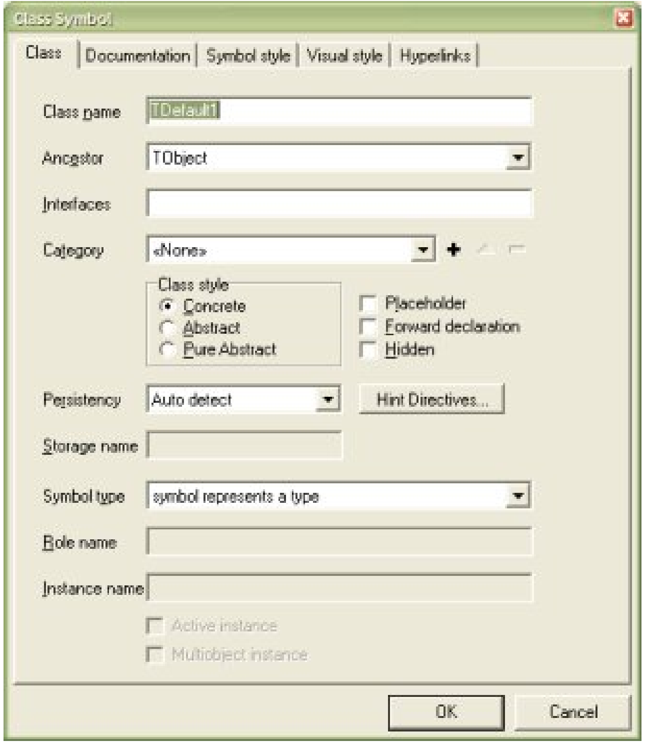


Figure 9 — part of the Class Diagram Editor toolbar.

These four buttons are used to add classes and interfaces to the diagram. The first button will create a new class, adding it to the model and the diagram — we‘ll discuss that in a moment. The second button allows you to choose a class that already exists in the model, and add it to the diagram. The third creates a new interface, adding it to both the model and the diagram while the fourth button will add an existing interface to the diagram. For this topic, we‘ll focus on the first button.

Click this first button (actually the fourth from the left). The cursor will change to indicate that you‘re adding a class.

Click on the diagram where you would like to add the class. The Class Symbol editor will be displayed (see figure 10).



Class Symbol

Class name: TDefault1

Ancestor: TObject

Interfaces:

Category: «None»

Class style: ● Concrete ○ Abstract ○ Pure Abstract

□ Placeholder □ Forward declaration □ Hidden

Persistency: Auto detect

Storage name:

Symbol type: symbol represents a type

Role name:

Instance name:

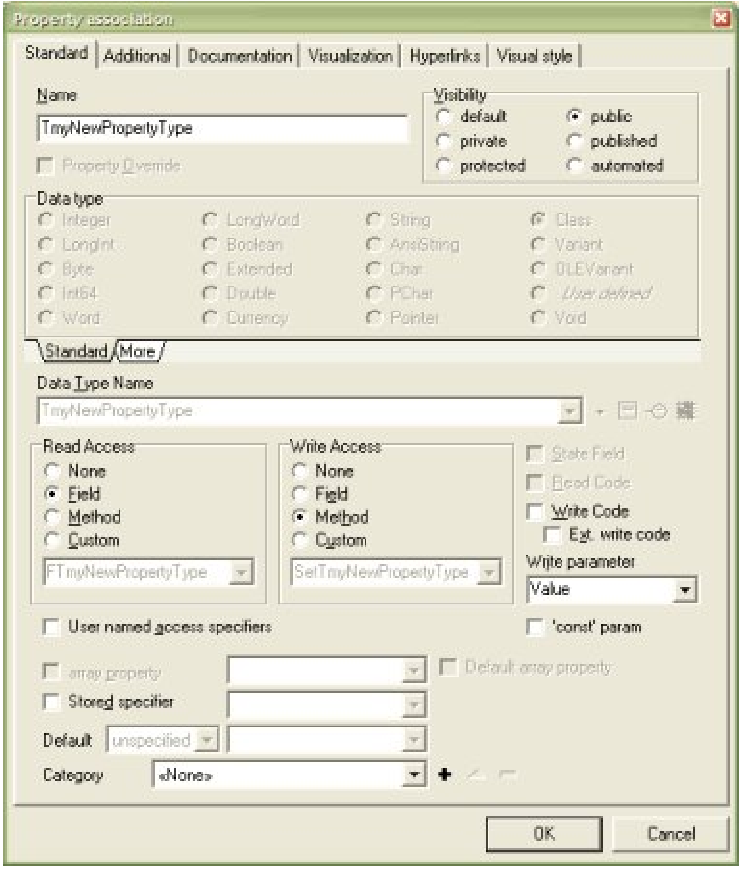
□ Active instance □ Multiobject instance

Figure 10 — the Class Symbol editor.

This editor allows you to define most every facet of a class — its role, appearance, documentation, etc. The Class Name field should be self-explanatory — enter your class‘ name here. Use the Ancestor drop-down to select an ancestor class. The other fields on this tab and the other tabs are optional; for now just set the first two fields and click OK. Your new class will appear in the diagram.

Press F3 (or select the Classes tab) to switch to the Classes View. If your diagram only contains the one class, drag some classes from the Classes View to the diagram to add them. Now notice that, as you select the various classes in your diagram, they become selected in the Classes View as well. In addition, their members are displayed in the Member List.

Your new class is rather empty, so let‘s add some properties. We need the Member List to add methods, events and simple properties and we‘ll be discussing that in Tutorial 5, but we can easily add class-type properties and fields in the Diagram Editor, so let‘s try that now. In the Class Diagram Editor, click the Add Property to Model tool button (8th from the left). Now click-and-hold in the class symbol to which you are adding the property, and drag to the class symbol whose type the property will be. For example, if you have a diagram containing the classes TmyClass and TmyNewPropertyType, to add a property to TmyClass of type TmyNewPropertyType, select Add Property to Model, then click on the TmyClass symbol and drag to the TmyNewPropertyType symbol. The Property Association dialog will appear (see figure 11).



Name: TmyNewPropertyType

□ Property Override

Visibility: ○ default ○ private ○ protected ● public ○ published ○ automated

Data type: ○ Integer ○ LongInt ○ Byte ○ Int64 ○ Word ○ LongWord ○ Boolean ○ Extended ○ Double ○ Currency ○ String ○ AnsiString ○ Char ○ PChar ○ Pointer ● Class ○ Variant ○ OLEVariant ○ *User defined* ○ Void

Data Type Name: TmyNewPropertyType

Read Access: ○ None ● Field ○ Method ○ Custom — FTmyNewPropertyType

Write Access: ○ None ○ Field ● Method ○ Custom — SetTmyNewPropertyType

□ State Field
□ Read Code
□ Write Code
□ Ext. write code

Write parameter: Value

□ User named access specifiers
□ 'const' param
□ array property
□ Default array property
□ Stored specifier

Default: unspecified

Category: «None»

Figure 11 — the Property Association dialog.

This is a complex dialog, but don‘t be daunted. Most the fields on this tab are the values you‘re used to entering in Delphi by hand: Name is the name of the new property, this defaults to the name of the properties class type. Use the Visibility group box to set the new property‘s visibility. The Data Type fields are disabled since, by adding a class-type property in the diagram, we already know the data type. The Read and Write Access group boxes allow you to set the methods of access for this new property. This is a very powerful and timesaving feature of ModelMaker; any accessor methods or state fields needed by this property are added automatically and are always kept up-to-date by ModelMaker. If you change the name of this property, ModelMaker will automatically update its accessors and/or state field.

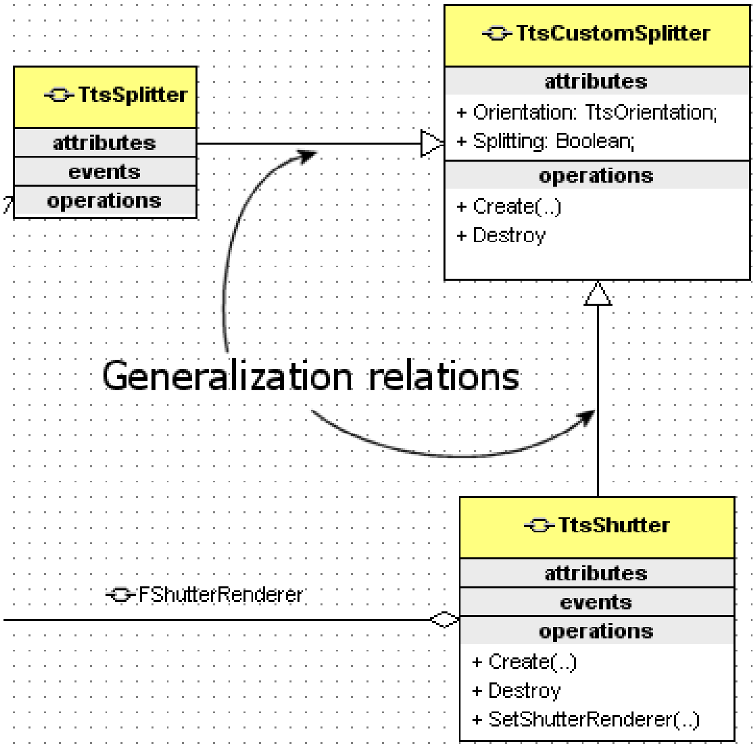


Figure 12 — Automatically visualized Generalization relations.

Changing Relations in a Diagram

You may have noticed that when visualizing your classes, if a class is added to a diagram and that class‘ ancestor is also in the diagram, the Generalization relation is automatically visualized (see figure 12).

There are a great many options and wizards for advanced handling and automation of visualizations, but the Generalization relation is pertinent to our current topic, so for our final thought here, notice the following:

You can change the ancestor of a class, not just within the diagram but also for the whole model, from within a class diagram. Make sure you are editing a class diagram that contains 3 classes, one of which is descendant of another in the diagram. Open the Classes View (so you can see the results) by pressing F3. Click the Generalization arrow to select it. A “grabber” should appear at each end of the arrow (see figure 13).

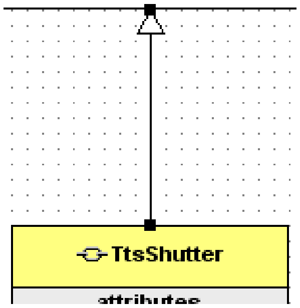


Figure 13 — A Generalization arrow, selected.

To change the class‘ ancestor, click the grabber at the arrowhead, and drag the arrowhead to a different class. Notice in the Classes View that the ancestor of your class has changed. This is a good example of the interconnectedness of the various editors — they are all looking into the same active code model. Note that this same method can be used to change the class type of a class‘ property within a diagram; drag the Property relation to a new class to change the property‘s type.

This has been an incomplete view of diagramming and the class creation process. See Tutorial 4 for an overview of diagramming in ModelMaker and Tutorial 5 for an overview of the class implementation process.

Return to the Introduction.
Go back to the previous tutorial.
Go on to the next tutorial.

Getting Started with ModelMaker

Tutorial 4 — Diagramming Overview

by Robert Leahey of Thoughtsmithy

Before we begin our discussion of diagrams, remember the following: ModelMaker maintains an internal active code model from which all source code units are generated. The various views (including the Diagram Editor) are simply different ways of looking into that model. The diagrams are not static, stand-alone images; they are graphical representations of the internal model, thus diagram symbols that represent code entities are actually showing us a representation of the entity in the model not just a graphic symbol. It‘s important to remember that distinction lest you try to delete a class from a diagram and wonder why the class still exists in the model.

ModelMaker currently offers the ability to create and manage 10 different types of diagrams, most of which are UML compliant. While most of the diagrams have toolbars and concepts that are specific to their types, there is some common ground with which I will acquaint you here.

There are two parts to the diagramming interface; the Diagrams View (F5) and the Diagram Editor (F8). The reason that these are mentioned separately is that it is possible to view the Diagram Editor while displaying any of the three main views (Classes, Units or Diagrams). See tutorial 2 for a listing of possible views and keystroke shortcuts. For the moment, display both diagramming views by pressing F5 (and if the Diagram Editor is not subsequently displayed, F8). If you are in a new project, with no diagrams, the Diagrams View should be blank and the Diagram Editor and toolbar should be grayed.

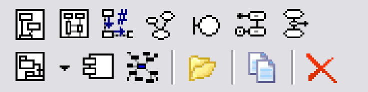


Figure 14 — the Diagrams View toolbar.

The Diagrams View toolbar (figure 14) contains buttons which will add each of the supported diagram types (Class, Sequence, Collaboration, Use Case, Robustness, State, Activity, Unit Dependency, Implementation and Mind Map).

To begin, let‘s add a new class diagram. Click the Add Class Diagram button (the upper-left button) in the Diagrams Views toolbar. Note that a new diagram is added to the Diagrams View tree and the diagram editor is activated with the new class diagram. You can rename the diagram by selecting its node and pressing F2. It‘s always possible to simply add a new diagram to the diagram list by clicking one of the “Add...” buttons. Try clicking some of the other “Add...” buttons now to add some other diagrams to the tree. Notice that as the various diagram types are added, the Diagram editors toolbar displays tool buttons specific to that diagram type. You also see some tool buttons that are common to all diagram types and we‘ll discuss these in a bit.

The Diagrams View is hierarchical; we can add some diagrams and rearrange them to be children of other diagrams. Right-click the Diagrams View and select Advanced Add... from the context menu. The resulting dialog allows you to specify a name for your diagram, select a diagram type and assign a parent diagram all in one step. Select one of your existing diagrams as the parent for your new creation and click OK. Note that the new diagram is added to the tree as the child of the existing diagram. This is a nice organizational feature and comes in handy in projects that contain numerous diagrams. For existing diagrams, it‘s possible to “re-parent" them within the Diagrams View by simply holding down the Ctrl key and dragging the diagram to its new parent.

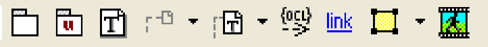


Figure 16.

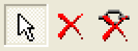


Figure 15.

Next we'll look at some of the design elements common to all diagrams. Select any of the diagrams you have added and have a look at the toolbar of the Diagram Editor. For any diagram you select, you should see figure 15 at the far left of the toolbar and figure 16 at the far right.

Moving from left to right, these common tools are as follows:

– Select. Use this tool to select diagram elements.

– Delete from Diagram. This tool will delete a symbol from a diagram, but will not delete the entity it represents from the model.

– Delete Symbol and Linked Entity. This tool will delete a symbol from the diagram and will also delete the entity it represents from the model.

– Add Package. Use this tool to create a package symbol in your diagram. This symbol is "container" and can hold other symbols, much like TPanel in Delphi.

– Add Unit Package. Similar to the Add Package tool, this tool adds a symbol representing a physical Delphi unit to your diagram. The symbol automatically displays the entities contained within the unit.

– Add Annotation. This tool will create a standard UML annotation symbol in which you can enter text of any sort.

– Add Documentation Link. This tool and the next, Add Linked Annotation, have some subtle differences, but they can be summed up as follows: the three options in this drop-down button allow you to connect a symbol and an existing annotation symbol. Whereas the next tool, Add Linked Annotation, will create a new annotation symbol and link it to the selected symbol. The three options in the drop-down menu allow you to add your own annotation text, display the linked entity's documentation or its One Liner comment.

– Add Linked Annotation. This tool, like the one above, links documentation to a diagram symbol. Using this tool will add a new annotation symbol and link it to the selected symbol. The three options in the drop-down menu allow you to add your own annotation text, display the linked entity's documentation or its One Liner comment.

– Add Constraint Relation. This is rather self explanatory; use this tool to add a constraint relation between two symbols to the diagram.

– Add Hyperlink. This is a powerful tool. This symbol, when added, will allow you to add a hyperlink to your diagram which can link to another diagram, a code model entity and/or an external document.

– Add shape. Use this tool to add simple geometric shapes to your diagram.

– Add Image. This tool allows you to add an image from an external file to your diagram.

When trying to alter the appearance of your diagrams, keep in mind that double-clicking on a diagram symbol (including associations and relations) will display that symbol‘s editing dialog where you can change the visual attributes of the diagram symbol. However, before you start trying to format your diagrams one symbol-at-a-time, keep in mind the following: ModelMaker provides access to the display styles of diagrams and their symbols using an inheritance model. It is possible to set styles for diagrams at an environment level via the Diagrams tab in the Environment Options dialog; to override these settings for a particular project, you would do so in the Project Options dialog, available from the Options menu. To override the settings for a diagram, double-click the diagram to display the Diagram Style dialog. Likewise, double-click a symbol to display its Symbol Style dialog.
This should provide enough of a basic overview of diagramming in ModelMaker to get you started. See the online help for information about specific diagram types.

Return to the Introduction.
Go back to the previous tutorial.
Go on to the next tutorial.

Getting Started with ModelMaker

Tutorial 5 — Implementing Classes/the Member List

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

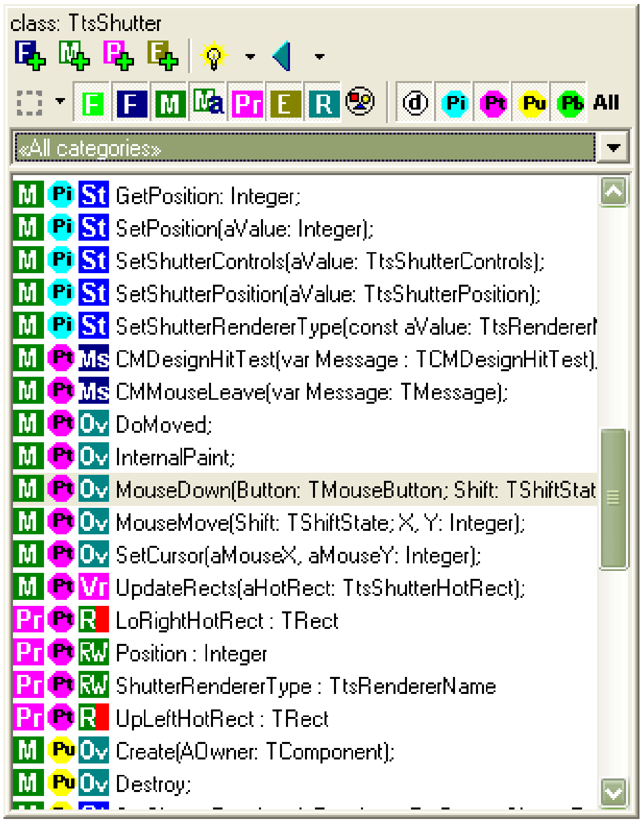
Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training



class: TtsShutter

«All categories»

M Pi St GetPosition: Integer;
M Pi St SetPosition(aValue: Integer);
M Pi St SetShutterControls(aValue: TtsShutterControls);
M Pi St SetShutterPosition(aValue: TtsShutterPosition);
M Pi St SetShutterRendererType(const aValue: TtsRendererN
M Pt Ms CMDesignHitTest(var Message : TCMDesignHitTest);
M Pt Ms CMMouseLeave(var Message: TMessage);
M Pt Ov DoMoved;
M Pt Ov InternalPaint;
M Pt Ov MouseDown(Button: TMouseButton; Shift: TShiftStat
M Pt Ov MouseMove(Shift: TShiftState; X, Y: Integer);
M Pt Ov SetCursor(aMouseX, aMouseY: Integer);
M Pt Vr UpdateRects(aHotRect: TtsShutterHotRect);
Pr Pt R LoRightHotRect : TRect
Pr Pt RW Position : Integer
Pr Pt RW ShutterRendererType : TtsRendererName
Pr Pt R UpLeftHotRect : TRect
M Pu Ov Create(AOwner: TComponent);
M Pu Ov Destroy;

Figure 17 — the Member List.

Given the wide range of options available in adding class members, this step is more informative than tutorial.

Ok, we’ve looked around enough — it’s time to get serious about implementing our classes.

The pane that you’ll need to get to know well is the Member List (see figure 17). This window is found below the triumvirate of the Classes View/Units View/Diagrams View.

The Members List allows you to manage the members (fields, methods, properties and events) of your classes. Note that there are many options that control the appearance of the Member List so my screen shots may not look the same as the Member List in your copy of ModelMaker.

The first thing to notice is that the Member List is just that — a list of a class‘ members. Remember that ModelMaker keeps an active model of your code internally and that all of the various editors are simply different views into the model; the Member List is another example of that. Any changes you make here are simply changing the model, not the physical source code files (unless you have Auto Generation Enabled) until you generate the source from the model.

On the General tab of the Environment Options dialog (available from the Options menu) is a group box entitled Members Appearance. If you check the first three items, Display Type Bitmaps, Display Visibility Bitmaps and Display Info Bitmaps, you will see all three columns of graphics in your Member List (as shown in figure 17). Select all three options and we’ll have a look at the various icons presented in the Member List.

The first two columns in the Member List represent the member type and visibility:

F — Field	d — default visibility
M — Method	Pi — private
Pr — Property	Pt — protected
E — Event	Pu — public
R — Method Resolution Clause	Pb — published

The third column displays a contextual icon depending on the member type. For methods, the icons represent bindings as follows:

St — static	Vr — virtual	Ov — override
Ab — abstract	Dy — dynamic	Ms — message

If the member is a property, the icons are as follows:

RW — Read/write	R — Read-only	No — Property override

The Member List allows you to add and edit members as well. Here we will take a general look at that process.

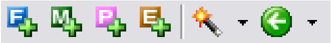


Figure 18 — Add Members toolbar.

Near the top of the Member List, you should see the toolbar in figure 18. The first four buttons on this toolbar are for adding members; they are, from left to right, Add Field, Add Method, Add Property and Add Event. Clicking one of these buttons will display the appropriate editor for the memeber type you selected. You should notice some similarities in each of the editors: Name, Visibility and Type. Also common to each editor is the Documentation tab where you can add documentation for your new member. The documentation you add is attached to the member within ModelMaker‘s internal model and can be used in a variety of ways. For example, you can emit the documentation with your code as comments (see Tutorial 7) or link to it within a diagram as an annotation (see Add Documentation Link and Add Linked Annotation in Tutorial 4). It is also available via the ModelMaker OpenTools API (see Tutorial 10).

In addition to the common elements, each editor contains settings specific to its member type. For instance, in the Property Editor, you can control the property‘s read and write access while the Event Editor allows you to generate a dispatch method.

Here are some quick thoughts on each of the member types which should help you to get more comfortable with ModelMaker:

Fields and Methods

If you‘re trying to add a propert‘s state field or an access method, stop. One of the key advantages of ModelMaker is its automation, and the automation surrounding the creation of properties is high. See the next section for more information.

Make sure you visit the ModelMaker online help by pressing F1 in the editor dialogs. Some items of note are the Owned and Initialized options for fields and the Inheritance Restricted option for methods.

Properties

As mentioned in the previous section, ModelMaker can greatly simplify the process of adding a property through its automation features. This is one of the reasons many people choose to work in ModelMaker as their code editor (as a Delphi IDE replacement); when adding a property by hand in Delphi, you write the declaration code for the property, its access methods, perhaps a state field and the implementation code for the access methods. When adding a proprty in ModelMaker, you add the property. Period.

Within the Property Editor you will find options for specifying the read/write access for the property (state field, method or none) as well as for generating some rudimentary read/write access code. Once you have set these options and clicked OK, ModelMaker adds not only the property, but the related access code. The generated code is “owned” by the property — if you change the name of the property or delete it, all the generated code is automatically renamed or deleted as well. Try experimenting with these settings and read in the online help for more information.

Events

Adding an event is somewhat similar to adding a property; most of the overhead is managed for you by ModelMaker. In the Event Editor dialog, you select a name, event type (event types are managed in the Event Type Library, available in the Events tab) and visibility. You can also specify a dispatch method by checking the Dispatch checkbox. The generated method will be owned by the event similar to the way a property owns its access methods.

Note that if you wish your dispatch methods to automatically be prefixed (with “Do”, for example) you can specify this and other coding style options in the *Coding Style* tab of the *Project Options* dialog under the *Options* menu.

Return to the Introduction.
Go back to the previous tutorial.
Go on to the next tutorial.

Getting Started with ModelMaker

Tutorial 6 — Delphi Integration

by Robert Leahey of Thoughtsmithy

While some developers use ModelMaker as a complete Delphi IDE replacement, most seem to prefer to use ModelMaker and Delphi simultaneously — switching back and forth as needed. To support this, ModelMaker offers extensive Delphi integration features, which we will examine here.

Delphi Controls ModelMaker

ModelMaker installs a Delphi integration expert into the Delphi IDE, which allows you to control ModelMaker somewhat from within Delphi.

First, start Delphi if it is not running, and have a look at the ModelMaker menu in the main Delphi menu bar. Notice that the first menu item allows you to launch ModelMaker. Also notice that most of the items in the ModelMaker menu require ModelMaker to be running — if it is not, those items will be disabled.

Launch ModelMaker now, if it is not running, by selecting the first menu item.

Warning: *in order to avoid overwriting some of your production code, make a copy of the unit or units you are going to import before continuing.*

The first thing we'll do is to create a new project and import into it some existing classes, all from within Delphi. Open, in Delphi, a project that you'd like to import into ModelMaker. From the ModelMaker menu, select *Convert Project to Model*. The ModelMaker window should come to the front, most likely asking whether to save your current model. After you answer that dialog as appropriate, ModelMaker will create a new project and import the Delphi project's units.

Some of the other menu items that operate similarly are *Add to Model*, which causes the active unit in Delphi to be imported into the current model in ModelMaker; *Add Files to Model*, which allows you to select units to be added to the current model in ModelMaker and *Convert to Model*, which will create a new model in ModelMaker and import the active Delphi unit into it.

Two other noteworthy menu items in the Delphi ModelMaker menu are *Jump to ModelMaker* and *Refresh in Model*. *Jump to ModelMaker* will bring ModelMaker to the front and attempt to find the current code block within the model. *Refresh in Model* will cause ModelMaker to re-import the active unit into the current model.

ModelMaker Controls Delphi

If you find that you like editing your code in ModelMaker, and prefer to have ModelMaker control Delphi, you have that option. Make sure that you are familiar with the process of generating code from within ModelMaker, by reading tutorial 1, then examine the Delphi portion of ModelMaker's main toolbar (see figure 19).

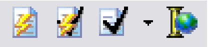


Figure 19 — Delphi control buttons.

The first button, *Generate*, you should already be familiar with.

The second button, *Generate and Perform Default Compile Action*, will cause the current unit to be generated, and will request that Delphi perform one of three actions: Syntax Check, Compile or Build. This default compile action can be set on the next button's drop-down menu.

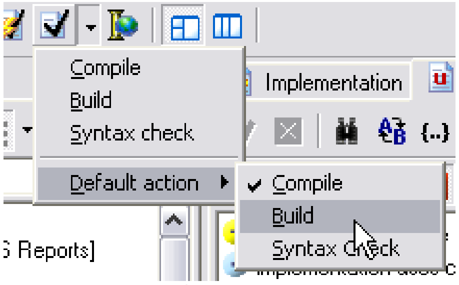


Figure 20 — Default Compile Action drop-down menu.

The third button (its hint will vary depending on the default compile action), will cause Delphi to perform the currently selected default action. This default is set in this button's drop-down menu (see figure 20); after setting a default, that action will be taken any time this button or the previous button is clicked. Note: these Delphi actions will be enacted upon whatever the currently loaded Delphi project may be.

The fourth button, *Locate in Delphi*, will bring Delphi to the front and attempt to locate the current code block in the code editor. It will open the current unit in Delphi if necessary.

Two-way Integration

The most complete integration can be achieved through ModelMaker's two-way integration, although a warning must be issued here: normally ModelMaker works with a safety net of allowing you to determine when code will be generated and when it will be refreshed into the model. Even if you enable auto-generation (see tutorial 1), you still have control over when code is re-imported if it has been changed outside of ModelMaker. If you enable the two-way integration, you will lose this control and code will automatically be generated and refreshed without warning. If there are changes to the code you do not wish to overwrite without warning, you should be careful using this option.

Enabling this feature basically causes ModelMaker to generate your units any time they change in ModelMaker and to re-import them any time they change outside of ModelMaker. To use this feature, do the following:

In ModelMaker, unlock code generation. Select one or more units to auto-generate and select the *Enable Auto-Generation* button for each.

Select *Environment Options* from the *Options* menu in ModelMaker. In the General tab, under Code Generation and Import, check the *Auto Refresh from IDE* and *Refresh Keeps Units Enabled* check boxes.

In Delphi, select *Integration Options* from the ModelMaker menu. In that dialog, select *Enable Auto Refresh*.

Open one of your auto-generated units in Delphi by clicking ModelMaker's Locate in Delphi button. If you want, arrange your ModelMaker and Delphi code editor windows so that you can see both at the same time. Now go back to ModelMaker and change something in that unit, then save the model. You should see the unit update in Delphi automatically. Now change something in the unit in the Delphi code editor, then save the unit. The unit will be refreshed in ModelMaker. Viola! Two-way code editing integration.

Another warning: if your code comment importing options are not correctly set up, you may see "comment creep" or worse, you might see comments deleted without warning. Be very careful using this two-way feature until you understand how to set up code comment importing options. For more information on that, see the next tutorial.

Return to the Introduction.
Go back to the previous tutorial.
Go on to the next tutorial.

home | software development | training services | print | audio productions | lego sculptures | contact us

Getting Started with ModelMaker

Tutorial 7 — Commenting Code / Macros

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training

The text for this tutorial is not yet complete.

home | software development | training services | print | audio productions | lego sculptures | contact us

Getting Started with ModelMaker

Tutorial 8 — Differencing

by Robert Leahey of Thoughtsmithy

The text for this tutorial is not yet complete.

Getting Started with ModelMaker

Tutorial 9 — The Unit Code Editor

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training

The text for this tutorial is not yet complete.

THOUGHT SMITHY

home | software development | training services | print | audio productions | lego sculptures | contact us

Getting Started with ModelMaker

Tutorial 10 — Advanced Stuff

by Robert Leahey of Thoughtsmithy

Introduction

Tutorial 1

Tutorial 2

Tutorial 3

Tutorial 4

Tutorial 5

Tutorial 6

Tutorial 7

Tutorial 8

Tutorial 9

Tutorial 10

ModelMaker Training

The text for this tutorial is not yet complete.